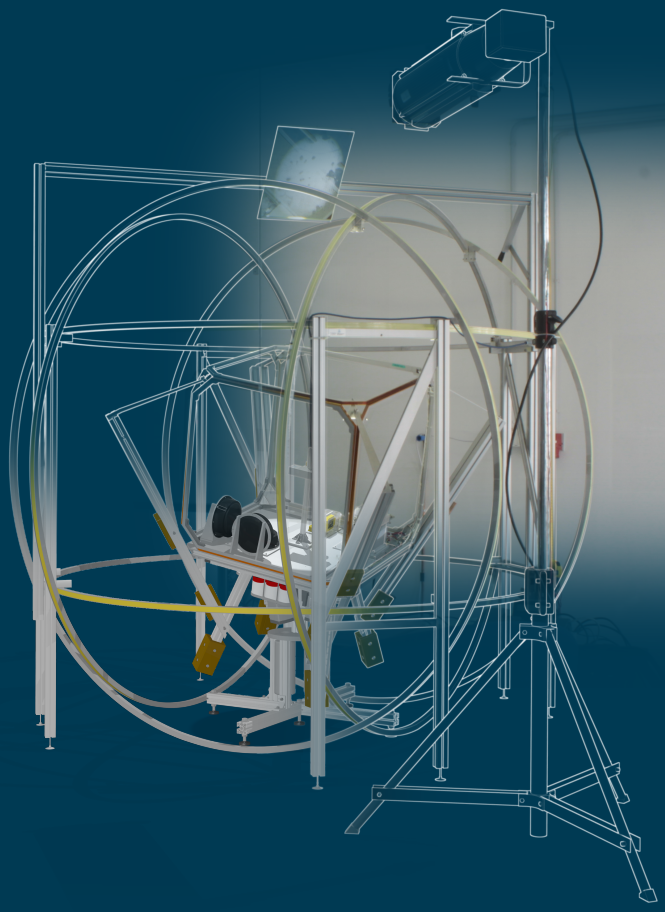
Michael Dumke

Satellite attitude control system for demonstration purposes

Diploma Thesis R 1138 D

September 2011



Prof. Dr.-Ing. Peter Vörsmann,
Dr.-Ing. Carsten Wiedemann
Technische Universität Braunschweig
Institut für Luft- und Raumfahrtsysteme
Hermann-Blenk-Straße 23, 38108 Braunschweig

Dr.-Ing. Stephan Theil,
Dipl.-Ing. Ansgar Heidecker
Deutsches Zentrum für Luft- und Raumfahrt
Institut für Raumfahrtsysteme
Robert-Hooke-Straße. 7, 28359 Bremen

Pages: 202
Figures: 60
Tables: 8

Technische Universität Braunschweig

Institut für Luft- und Raumfahrtsysteme

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die nachfolgende Arbeit selbstständig und nur unter Zuhilfenahme der angegebenen Literatur angefertigt habe.

Michael Dumke

Bremen, den 29. September 2011

The DLR Institute of Space Systems located in Bremen commissioned a new laboratory for experiments concerning the attitude control of spacecrafts. The laboratory was named *Facility for Attitude Control Experiments (FACE)*, and the necessary equipment and hardware was acquired. The subject of this diploma thesis is bringing the new facility into service and demonstrating its capabilities by means of realistic example experiments inspired by actual space missions.

The main component of the facility is an air bearing table for simulating zero gravity with respect to the rotational degrees of freedom. An experimental satellite mounted onto the table is able to rotate around all its major axes with only minimal friction between the air bearing parts. Available hardware is used to build an attitude control system for the inertial measurement and adjustment of the satellite's attitude. For this, the on-board communication between the sensors, the actuators, and the on-board computer are implemented and put into operation.

For processing the sensor signals and commanding the actuators, reusable software interfaces are designed for user-friendly interaction within the MATLAB™/Simulink™ development environment. A basic subsystem for *attitude determination* and *attitude control* is implemented with the developed instruments, demonstrating the overall design process employed for the FACE.

Contents

Nomenclature

Attitude determination and control

$\underline{\underline{A}}$	state matrix
$\underline{\underline{B}}$	input matrix
$\underline{\underline{C}}$	output matrix
$\underline{\underline{D}}$	feedthrough matrix
Δ	sensor value resolution
η	random walk of a fiber optical gyroscope
$\underline{\underline{F}}$	Jacobian matrix of a state space with respect to the state vector
$\underline{\underline{G}}$	state noise coupling matrix
$\underline{\underline{H}}$	Jacobian matrix of a measurement equation with respect to the state vector
$\underline{\underline{K}}$	controller gain matrix
$\underline{\underline{P}}_0$	initial condition for the state covariance matrix
$\underline{\underline{Q}}$	system noise covariance matrix (Kalman), weighting matrix of the state vector deviation (linear quadratic regulator (LQR))
$\underline{\underline{R}}$	measurement noise covariance matrix (Kalman), weighting matrix of the control vector deviation (LQR)
σ	standard deviation
σ^2	variance
$\underline{u}$	control signal
$\underline{w}, \underline{v}$	normally distributed noise
$\underline{x}, \underline{\dot{x}}$	state vector, derivative of the state vector with respect to time
$\underline{x}_0$	initial condition for the state vector
$\underline{z}$	measurement vector

Celestial mechanics

a	semi-major axis of a Kepler orbit
b	semi-minor axis of a Kepler orbit
e	eccentricity of a Kepler orbit
γ	gravitational constant
i	inclination
Ω	longitude of the ascending node
ω	argument of periapsis
p	parameter of a Kepler orbit
Θ	true anomaly of a Kepler orbit
U_A, u_A	potential energy of point mass A and specific potential energy

Υ	direction to the vernal equinox

Frames

a	assembly station frame (see appendix A.3)
b	body fixed frame (see appendix A.2)
e	Earth centered earth fixed (ECEF) frame (see appendix A.4)
i	inertial frame, or Earth centered inertial (ECI) frame (see appendix A.4)
o	orbital frame (see appendix A.1)
r	reference frame at a c. m., aligned with an inertial frame
x_a, y_a, z_a	the ortogonal unit vectors of frame a (right-hand rule)

Rigid body mechanics

$\underline{a}_A^c$	acceleration of point A in coordinates of system c
$\underline{B}_{\text{Earth}}^c$	magnetic field vector of the Earth in coordinates of system c
$\underline{D}_{\text{coil}}^c$	dipole moment of a coil in coordinates of system c
$\underline{e}$	integrated quaternion control deviation
$\underline{\varepsilon}_{b,a}^c, \varepsilon_{b,a}^c, \overline{\varepsilon}$	error angle of frame a with respect to frame b in coordinates of system c, its absolute value, mean error angle
$\underline{\epsilon}_{b,a}^c, \epsilon_{b,a}^c, \overline{\epsilon}$	error rate of frame a with respect to frame b in coordinates of system c, its absolute value, mean error rate
ε	rotation angle of a quaternion
$\underline{F}_a^c$	force in frame a in coordinates of system c
$\underline{g}_A^c$	gravitational acceleration of point mass A in coordinates of system c
$\underline{h}_{b,\mathbf{A}}^c, h_{b,\mathbf{A}}^c$	angular momentum of body $\mathbf{A}$ with respect to frame b in coordinates of system c and its absolute value
$\underline{h}_{b,A}^c, h_{b,A}^c$	angular momentum of point mass A with respect to frame b in coordinates of system c and its abolute value
$\underline{H}_{b,A}^c, H_{b,A}^c$	specific angular momentum of point mass A with respect to frame b in coordinates of system c and its abolute value
$\underline{\dot{h}}_{b,\mathbf{A}}^c, \dot{h}_{b,\mathbf{A}}^c$	change in angular momentum over time of body $\mathbf{A}$ with respect to frame b in coordinates of system c and its abolute value
$\underline{\underline{I}}_{\mathbf{A}}^c$	moment of inertia tensor of body $\mathbf{A}$ in coordinates of system c
i, j, k	imaginary components of a quaternion if descriped in form of a summation
m_A	mass of point mass A
$m_{\mathbf{A}}$	mass of body $\mathbf{A}$
$\underline{n}$	rotation axis of a quaternion

$\underline{\omega}_{b,a}^{c}, \omega_{b,a}^{c}$	angular velocity of frame a with respect to frame b in coordinates of system c and its abolute value
$\underline{\underline{\omega}}_{b,a}^{c}$	skew-symmetric form of angular velocity of frame a with respect to frame b in coordinates of system c
$\underline{p}_{b,A}^{c}, p_{b,A}^{c}$	linear momentum of point mass A with respect to frame b in coordinates of system c and its abolute value
$\mathbf{q}$	quaternion
$\mathbf{q}_{b}^{a}$	transformation quaternion of frame a with respect to b
$\underline{q}, \underline{r}, \underline{s}$	basis vectors
$\underline{r}_{b,a}^{c}, r_{b,a}^{c}$	radius vector of frame a with respect to frame b in coordinates of system c and its abolute value
$\underline{r}_{b,A}^{c}, r_{b,A}^{c}$	radius vector of point A with respect to frame b in coordinates of system c and its abolute value
$\underline{T}_{b,a}^{c}, T_{b,a}^{c}$	torque of frame a with respect to frame b in coordinates of system c and its abolute value
t	time, specific point in time, time delay
$\underline{T}_{b,A}^{c}, T_{b,A}^{c}$	torque of point A with respect to frame b in coordinates of system c and its abolute value
$\underline{\underline{T}}_{b}^{a}$	transformation matrix of frame a with respect to b
$\underline{v}_{b,a}^{c}, v_{b,a}^{c}$	velocity of frame a with respect to frame b in coordinates of system c and its abolute value

1 Introduction

The DLR Institute of Space Systems located in Bremen analyzes and develops concepts regarding new technologies in space flight in a broad range of subjects. From new ideas of commercial space travel to the development of payloads for the scientific exploration of the Solar System. The department *Navigation and Control Systems*, a unit of Bremen's DLR branch, commissioned a new laboratory for the experimental simulation of *attitude control systems* to research new approaches in *attitude control* and *attitude determination*. The verification of newly developed hardware and the verification of computer simulation results are other examples for the utilization of the *Facility for Attitude Control Experiments*.

The test facility was built by the company *Astro- und Feinwerktechnik Adlershof GmbH* located in Berlin. The central element is an *air bearing* with the distinguishing feature of very low friction between both parts of the bearing. Loaded onto the bearing is an assembly station for the fixation of hardware components of an experimental satellite. This satellite is decoupled as far as possible from the laboratory's environment. A dedicated battery-based on-board power supply ensures long experiment durations for a realistic simulation of the attitude dynamics. This facility provides a disturbance free environment for the attitude control hardware that is able to work similar to a real free-floating spacecraft. In addition to the simulations of the rotational degrees of freedom, selected environmental influences can be employed, such as the Earth's magnetic field and the illumination by the Sun.

The subject of this diploma thesis is bringing the facility into service and demonstrating the capabilities of the test platform with scenarios based on realistic requirements for satellite mission. The fundamentals required for this thesis are discussed in a separate chapter. A short derivation introduces the celestial mechanics based on Kepler's laws of planetary motion, and a short insight is given into the rigid body mechanics, focusing on the simulation of the spacecraft's dynamics.

The laboratory is set up and put into operation according to the technical documentation for the test facility. This includes familiarizing with the mandatory operational procedures for the correct handling of all critical devices, such as the sensitive air bearing, and the high pressure lamp of the Sun simulator.

A very complex task is the implementation of the on-board communication between the on-board computer and the attitude control system's hardware. The basic standards for the most common serial interfaces is summarized with regards to the utilization of serial devices, and the encoding of user data in the transmission stream. Aside from the on-board computer, the available devices include fiber optic gyroscopes, reaction wheels, and magnetometers. Each device is put into operation separately. Initially this happens in a contained environment in which the wiring is carefully tested, in particular the wiring for the power supply. After the correct operation is verified, the communication proto-

col of each device is programmed according to the corresponding documentation. The implementations must be tested for a reliable communication and afterwards a software interface is built in order to ensure a user-friendly and reusable interface to the hardware devices.

Following the phase in the protected environment is the phase of integrating the hardware on the experimental satellite on top of the air bearing table. After the simultaneous usage of the newly developed software interfaces has been proven to work, initial tests with the attitude control hardware can be conducted. These include small slew maneuvers on a stable assembly station, meaning a center of mass below the pivot point of the air bearing.

The operational hardware is then used in two processes that have to be developed. Small movable weights controlled by stepper motors are attached to the assembly station in order to precisely adjust the center of mass of the test platform. The alignment of the center of mass and the pivot point of the air bearing happens autonomously in order to reduce the effort for this reoccurring task, and to gain a more accurate result. The second process estimates the moment of inertia of the experimental satellite by processing measurement data from the satellite's response to an induced disturbance torque. This estimate is important for the development of an attitude control system for an agile and precise control of the satellite's attitude.

Fiber optic gyroscopes measure the angular velocity of the experimental satellite. *Kalman* filters process the noisy sensor signals for estimating the attitude, purely relying on the inertially measured angular rate. The filtering also includes the compensation for the Earth's rotation that, if left unconsidered, would cause a drift in the reference attitude. The accurate attitude determination is necessary for following a guidance trajectory with a closed-loop control system, optimized on simplified satellite dynamics. The employed robust controller design does not suffice for the desired guiding precision during slew maneuvers, whereas the additional implementation of a feed forward controller greatly improves the ability to follow complex guidance maneuvers.

Preliminarily tests show the correct operation of the integrated hardware and the implemented algorithms on board the experimental satellite. The employed development environment MATLAB™ and the numerical simulation add-on Simulink™ are used for an efficient implementation process, from first prototypes to complex attitude control systems. The resulting real-time application for the on-board computer is monitored in real-time from a control computer in the laboratory, ensuring a safe and correct course of an experiment.

More complex scenarios demonstrate the reliability of the test assembly in order to conduct experiments with repeatable and comparable results. The demonstration of a ground station guidance mission shows the current state of the facility to underline its potential when employing more advanced algorithms and additional hardware.

The last chapter of this thesis is intended to support a user of the facility in their activity of implementing new algorithms, or integrating new hardware on the experimental satellite, giving the reader the opportunity for a faster and more efficient process based on the experience gained during the set up of the laboratory.

2 Fundamentals

This chapter discusses the fundamentals that are essential to understanding the rest of this thesis. The first part addresses the mechanics of the satellite's motion that is needed to understand basic orbit mechanics, different sensor types and the principles of actuators influencing the attitude of a satellite.

The second part of this chapter gives an overview of common digital interfaces for the communication on board of a satellite. The overview contains information about the wire connection between multiple devices, the electrical characteristics of signal levels, and transmission methods for a reliable communication.

2.1 Mechanics

The satellite's motion can be divided into two major parts. The first part is the translational motion, affecting the position around a central mass. The differential equations are important for the simulation and propagation of such an orbit around a central body and are also very useful for planning mission scenarios and guiding orbit maneuvers to a successful outcome. The second part is the rotational motion around the center of mass which, in most cases, can be considered as decoupled from the translational degrees of freedom. The description of this mathematical problem is needed for attitude control system of satellites as well as for simulation purposes.

2.1.1 Celestial mechanics

Celestial mechanics describes the motion of suns, planets, moons and artificial objects around each other. With just a few simplifications, a very accurate model can be built that is good enough to estimate first results in the analysis of space missions. The mathematical description starts with the momentum p of object A in an inertial reference system i. In classical mechanics, momentum is the product of the object's mass m and its velocity v.

$$\underline{p}^i_{i,A} = m_A \frac{\mathrm{d}_i \underline{r}^i_{i,A}}{\mathrm{d}t} = m_A \underline{v}^i_{i,A} \tag{2.1}$$

The superscript denotes the coordinate system of a physical quantity. The inertial change of the object's location over time $\frac{\mathrm{d}_i \underline{r}^i_{i,A}}{\mathrm{d}t}$ is of course the velocity. The formula is valid in three dimensional space, therefore an underlined symbol points to its vector characteristics.

Newton's second law of motion states the conservation of momentum. Unless a force acts upon the object A the momentum remains unchanged. If the mass of A is constant,

a change in momentum changes the velocity and accelerates the object.

$$\frac{\mathrm{d}_i \underline{p}^i_{i,A}}{\mathrm{d}t} = m_A \frac{\mathrm{d}_i \underline{v}^i_{i,A}}{\mathrm{d}t} = m_A \underline{a}^i_A = \underline{F}^i_A \tag{2.2}$$

This consideration has to be done in the inertial space i, i. e., a reference frame that neither accelerates nor rotates [1]. A common simplification is to treat the reference frame at the center of the Earth as inertial, which of course is not fully accurate.

To simulate the behavior of an object influenced by forces, equation 2.2 suffices to solve the problem. The acceleration $\underline{a}$ equals the summation of all acting forces divided by the object's mass.

$$\begin{aligned} \underline{a}^i_A &= \frac{1}{m_A} \sum_N \underline{F}^i_N \\ &= \frac{1}{m_A} \left(\underline{F}^i_{A,\text{ gravity}} + \underline{F}^i_{A,\text{ aerodynamic}} + \underline{F}^i_{A,\text{ solar radiation pressure}} + \cdots \right) \end{aligned} \tag{2.3}$$

Integrating the acceleration twice over time yields the position of the object. Numerically this problem is solved for known disturbance forces, but for an analytical expression the problem has to be reduced.

The first simplification in this derivation is the assumption of an environment free of disturbances, with the only force acting on the object being the gravitational force. For gravitational forces a second object B is needed. From now on, B will denote the central body of this two-body system. A second simplification is the assumption that B is much heavier than object A and B does not move at all. The simplest model of the gravitational force between two bodies is

$$\underline{F}^i_{A,\text{ gravity}} = \underline{g}^i_A = -\gamma \frac{m_B m_A}{{r^i_{B,A}}^3} \underline{r}^i_{B,A} = -\underline{g}^i_B \,, \tag{2.4}$$

where $\underline{r}^i_{B,A}$ is the displacement vector of A with respect to B's position in inertial coordinates. The EUCLIDEAN norm of a vector is represented by the same symbol without the underline, and γ is the gravitational constant [2].

$$\underline{r}^i_{B,A} = \underline{r}^i_{i,A} - \underline{r}^i_{i,B} \qquad r^i_{B,A} = |\underline{r}^i_{B,A}| \tag{2.5}$$

This gravity model is of course well known [3, 2]. Gravity is a conservative force field in which the object A, e. g., a satellite, trades its potential energy with its kinetic energy, like a pendulum. A conservative force field is characterized by the losslessness of this trade.

Here the potential energy U_A is defined as the energy necessary to completely remove the satellite A from this potential, that is, move the satellite to an infinite distance from the central body.

$$U_A = \int_{r^i_{B,A}}^{\infty} g^i_A \mathrm{d}r = \underbrace{\lim_{r^i_{B,A} \to \infty} \gamma \frac{m_B m_A}{r^i_{B,A}}}_{\to 0} - \gamma \frac{m_B m_A}{r^i_{B,A}} = -\gamma \frac{m_B m_A}{r^i_{B,A}} \tag{2.6}$$

As said earlier, the energy stored in the system does not change over time. The specific kinetic energy $\frac{1}{2}{v^i_{B,A}}^2$ and the specific potential energy $u_A = \frac{U_A}{m_A}$ is constant, as long as no other disturbance force exists and the satellite's mass is negligible compared to the central body.

Two particular cases deserve special attention. First the case with no potential energy ($r \to \infty$) and second the radius where the velocity is zero. To this point it is not known which radius results in a velocity of zero, but as of now this special radius is arbitrarily called $2a$, which will later simplify the resulting equations:

$$\frac{1}{2}{v^i_{B,A}}^2 - \gamma\frac{m_B}{r^i_{B,A}} = \text{const} = \frac{1}{2}{v^i_{B,A}}^2\Big|_{r^i_{B,A}\to\infty} = -\gamma\frac{m_B}{2a}\Big|_{v^i_{B,A}\to 0} \tag{2.7}$$

Solving this for the velocity results in the *vis-viva* equation [3, 4]:

$${v^i_{B,A}}^2 = \gamma m_B\left(\frac{2}{r^i_{B,A}} - \frac{1}{a}\right) \tag{2.8}$$

With equation 2.8 and the special variable a, for every radius $r^i_{B,A}$ of the satellite A, its velocity is computable.

Expecting a circular or elliptical motion of the satellite, the reference system is changed to polar coordinates with r as the radius and Θ for the circumferential direction. The absolute value of the velocity in polar coordinates can be computed with the *Pythagorean theorem* (cf. Fig. 2.1):

$${v^i_{B,A}}^2 = \left(\frac{\mathrm{d}_i r^i_{B,A}}{\mathrm{d}t}\right)^2 + \left(r^i_{B,A}\frac{\mathrm{d}_i\Theta}{\mathrm{d}t}\right)^2 \tag{2.9}$$

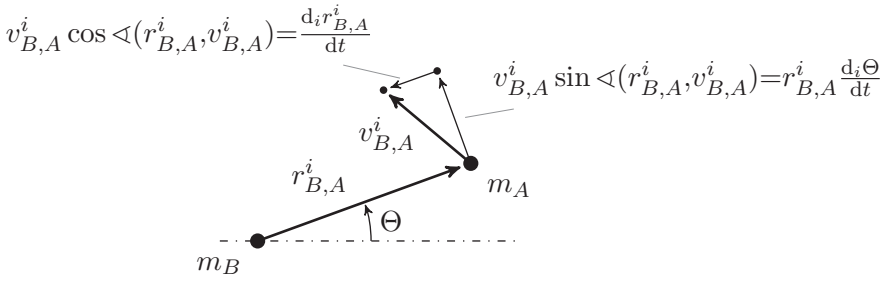


Figure 2.1: Geometric relations of the velocity in a polar coordinate system

The cross product of equation 2.1 and the radius vector r gives the angular momentum $\underline{h}$ of object A, with $\underline{H}$ being the specific angular momentum.

$$\underline{r}^i_{B,A} \times \underline{p}^i_{B,A} = m_A\underline{r}^i_{B,A} \times \underline{v}^i_{B,A} = \underline{h}^i_{B,A} = m_A\underline{H}^i_{B,A} \tag{2.10}$$

With the central body B being fixed on the origin, the angular momentum of A is constant, as is its energy. Rewriting the specific angular momentum with the help of the geometric relations from Fig. 2.1 leads to

$$H^i_{B,A} = |\underline{r}^i_{B,A} \times \underline{v}^i_{B,A}| = r^i_{B,A} v^i_{B,A} \sin \sphericalangle(r^i_{B,A}, v^i_{B,A}) = r^i_{B,A} r^i_{B,A} \frac{\mathrm{d}_i \Theta}{\mathrm{d}t}\,, \tag{2.11}$$

which can be rearranged to

$$\frac{H^i_{B,A}}{{r^i_{B,A}}^2} = \frac{\mathrm{d}_i \Theta}{\mathrm{d}t}\,. \tag{2.12}$$

Expansion of the radial velocity and substituting the inertial derivative of Θ with equation 2.12 leads to the expression

$$\frac{\mathrm{d}_i r^i_{B,A}}{\mathrm{d}t} = \frac{\mathrm{d}_i r^i_{B,A}}{\mathrm{d}_i \Theta} \frac{\mathrm{d}_i \Theta}{\mathrm{d}t} = \frac{\mathrm{d}_i r^i_{B,A}}{\mathrm{d}_i \Theta} \frac{H^i_{B,A}}{{r^i_{B,A}}^2}\,, \tag{2.13}$$

which is independent of the time.

The last step is inserting equation 2.9 into 2.7, and then making the formula independent of the time using equation 2.13:

$$\frac{1}{2}\left(\frac{\mathrm{d}_i r^i_{B,A}}{\mathrm{d}t}\right)^2 + \frac{1}{2}\left(r^i_{B,A} \frac{\mathrm{d}_i \Theta}{\mathrm{d}t}\right)^2 - \gamma \frac{m_B}{r^i_{B,A}} = -\gamma \frac{m_B}{2a} \tag{2.14}$$

$$\frac{1}{2}\left(\frac{\mathrm{d}_i r^i_{B,A}}{\mathrm{d}_i \Theta} \frac{H^i_{B,A}}{{r^i_{B,A}}^2}\right)^2 + \frac{1}{2}\left(\frac{H^i_{B,A}}{r^i_{B,A}}\right)^2 - \gamma \frac{m_B}{r^i_{B,A}} = -\gamma \frac{m_B}{2a} \tag{2.15}$$

Equation 2.15 is a first order differential equation. One solution for this differential equation is of course an ellipse, that conforms to *Kepler's laws of planetary motion*:

$$r^i_{B,A} = \frac{p}{1 + e \cos \Theta} \tag{2.16}$$

The graphical representation of this ellipse is given in Fig. 2.2. All parameters are shown in the graphic, including the arbitrarily chosen parameter a, which is the semi-major axis of the ellipse. The angle Θ in the polar coordinate system of an orbit is called *true anomaly* and the three parameters p (2.17), e (2.18) and b (2.19) from equation 2.16 are called *parameter*, *eccentricity* and the *semi-minor axis*.

$$p = \frac{{H^i_{B,A}}^2}{\gamma m_B} \tag{2.17}$$

$$e = \sqrt{1 - \frac{{H^i_{B,A}}^2}{a \gamma m_B}} \tag{2.18}$$

$$b = a\sqrt{1 - e^2} \tag{2.19}$$

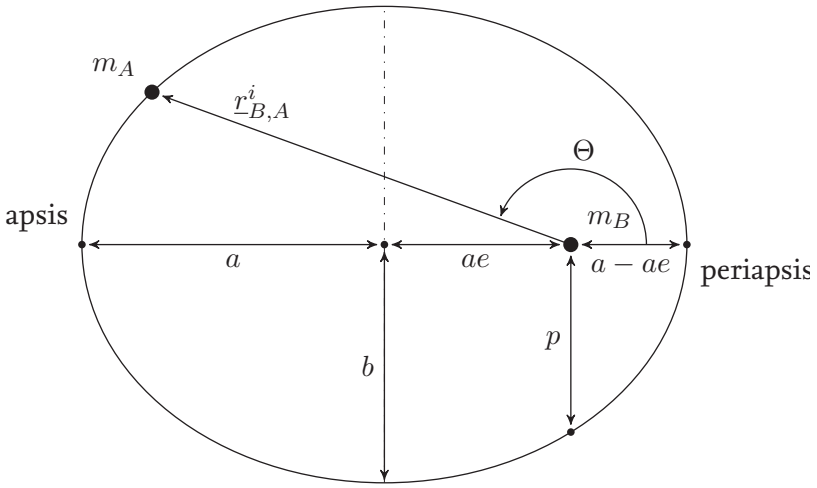


Figure 2.2: Parameters of an elliptical orbit

The parameter p and the semi-major a axis should not be confused with the momentum $p^i_{i,A}$ and the acceleration a^i_A. The two equations 2.8 and 2.16 are useful for many analyses of orbits where disturbances are very small. To use them in three dimensional space the orbit plane from Fig. 2.2 must be defined in relation to another reference plane, e. g., the equatorial plane of the Earth. All important elements of an orbit relative to a reference plane are depicted in Fig. 2.3.

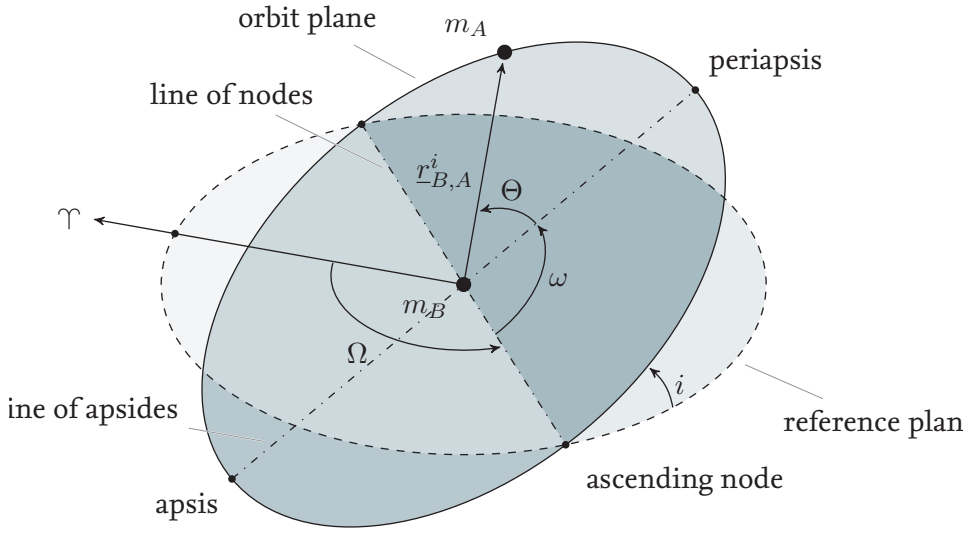


Figure 2.3: Parameters of an elliptical orbit relative to a reference plane

Three different angles describe this relation. Firstly, the *inclination* i tilts the planes with respect to each other and defines an intersection line called *line of nodes*. The second

angle rotates the orbit plane around the reference plane's normal direction. It is called *longitude of the ascending node* Ω and describes the angle between a reference direction Υ, e. g., the *vernal equinox*, and the point where the orbit crosses the reference plane in an upwards direction (*ascending node*), i. e., the direction of Θ is positive. The last angle is the *argument of periapsis* ω which rotates the orbit plane around its surface normal at the central body m_B.

The main equations 2.8, 2.16 and the parameters $(a, e, i, \omega, \Omega)$ are accurate enough to describe an orbit for early analyses. For the development of a concept for a satellite mission with basic guidance and attitude control, the assumptions of this model will generally suffice.

2.1.2 Rigid body mechanics

In this section the rotational degrees of freedom of a body will be analyzed. The body is called *rigid*, meaning it consists of multiple point masses that cannot move relative to each other, but only as a whole.

The angular momentum $\underline{h}$ is defined between two reference frames i and b. It is calculated using the cross product of the radius vector $\underline{r}$ from a reference location and the linear momentum $\underline{p}$ from equation 2.1.

The total angular momentum of the body $\mathbf{B}$ is the sum of all angular momenta taken over all point masses m_N.

$$\underline{h}^i_{i,\mathbf{B}} = \sum_N \left[\underline{r}^i_{i,N} \times \underline{p}^i_{i,N}\right] = \sum_N \left[\underline{r}^i_{i,N} \times (m_N \underline{v}^i_{i,N})\right] = \sum_N \left[m_N \underline{r}^i_{i,N} \times \underline{v}^i_{i,N}\right] \tag{2.20}$$

The inertial radius vectors $\underline{r}^i_{i,N}$ to each point mass can be split into one constant vector to a reference frame b and an individual radius vector relative to this frame. This change in reference frame is depicted in Fig. 2.4 for multiple point masses. For the derivative of

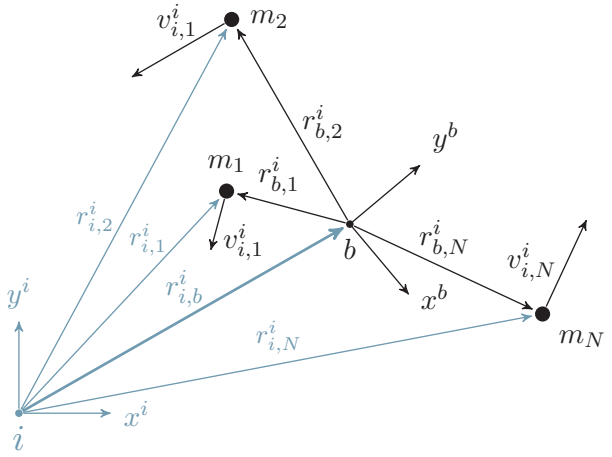


Figure 2.4: Total angular momentum

the inertial radius vector with respect to time, i. e., the velocity, the same splitting can be conducted.

$$\underline{r}^i_{i,N} = \underline{r}^i_{i,b} + \underline{r}^i_{b,N} \Rightarrow \frac{\mathrm{d}_i \underline{r}^i_{i,N}}{\mathrm{d}t} = \underbrace{\frac{\mathrm{d}_i \underline{r}^i_{i,b}}{\mathrm{d}t}}_{\underline{v}^i_{i,b}} + \frac{\mathrm{d}_i \underline{r}^i_{b,N}}{\mathrm{d}t} \tag{2.21}$$

The first addend of the derivative is identical for all point masses, and it is equal to the velocity $\underline{v}^i_{i,b}$ of the whole body in the inertial frame.

For the inertial derivative $\mathrm{d}_i/\mathrm{d}t$ of $\underline{r}^i_{b,N}$ that is relative to the body fixed frame b the Euler-transformation must be used. This transformation changes the derivative to the local frame $\mathrm{d}_b/\mathrm{d}t$ and also takes the rotation $\underline{\omega}^i_{i,b}$ between those two frames into account.

$$\frac{\mathrm{d}_i \underline{r}^i_{b,N}}{\mathrm{d}t} = \underbrace{\frac{\mathrm{d}_b \underline{r}^i_{b,N}}{\mathrm{d}t}}_{\overset{\text{rigid body}}{=}0} + \underline{\omega}^i_{i,b} \times \underline{r}^i_{b,N} \tag{2.22}$$

As a consequence of only considering rigid bodies, the derivatives of the radius vectors $\underline{r}^i_{b,N}$ are always zero in the body frame.

With equations 2.21 and 2.22 the total angular momentum of the body can be rearranged to

$$\underline{h}^i_{i,\mathbf{B}} = \sum_N \left[m_N \left(\underline{r}^i_{i,b} + \underline{r}^i_{b,N} \right) \times \left(\underline{v}^i_{i,b} + \underline{\omega}^i_{i,b} \times \underline{r}^i_{b,N} \right) \right] . \tag{2.23}$$

Rearranging this sum makes it possible to remove two summands, because they are equal to zero: By choosing the reference frame b on the center of mass (c. m.) of the body $\mathbf{B}$, the term $\sum_N \left[m_N \underline{r}^i_{b,N} \right]$ vanishes because this is exactly the definition of the c. m. [1].

$$\sum_N \left[m_N \underline{r}^i_{b,N} \times \underline{v}^i_{i,b} \right] = -\underline{v}^i_{i,b} \times \underbrace{\sum_N \left[m_N \underline{r}^i_{b,N} \right]}_{\overset{\text{c. m.}}{=}0} = 0 \tag{2.24}$$

$$\sum_N \left[m_N \underline{r}^i_{i,b} \times \left(\underline{\omega}^i_{i,b} \times \underline{r}^i_{b,N} \right) \right] = \underline{r}^i_{i,b} \times \left(\underline{\omega}^i_{i,b} \times \underbrace{\sum_N \left[m_N \underline{r}^i_{b,N} \right]}_{\overset{\text{c. m.}}{=}0} \right) = 0 \tag{2.25}$$

Without these terms the angular momentum of the body is left to be

$$\underline{h}^i_{i,\mathbf{B}} = \sum_N \left[m_N \underline{r}^i_{b,N} \times \left(\underline{\omega}^i_{i,b} \times \underline{r}^i_{b,N} \right) \right] + \sum_N \left[m_N \underline{r}^i_{i,b} \times \underline{v}^i_{i,b} \right] , \tag{2.26}$$

where the first addend being solely described in terms of the body fixed frame b and the body's rigid characteristics, thus the term can be split up into two parts, a geometrical part and the angular rate. This yields the abbreviation

$$\underline{\underline{I}}^i_{\mathbf{B}} \underline{\omega}^i_{i,b} := \sum_N \left[m_N \underline{r}^i_{b,N} \times \left(\underline{\omega}^i_{i,b} \times \underline{r}^i_{b,N} \right) \right] , \tag{2.27}$$

where $\underline{\underline{I}}$ is called *moment of inertia tensor* which has a matrix representation of the form 3×3. The matrix form of the tensor is constant in body fixed coordinates, due to the limitations of rigid bodies, which was the motive to change the frame in the first place.

The angular momentum of a body in a shorter form than equation 2.26 is

$$\underline{h}^i_{i,\mathbf{B}} = \underline{\underline{I}}^i_{\mathbf{B}}\underline{\omega}^i_{i,b} + m_{\mathbf{B}}\underline{r}^i_{i,b} \times \underline{v}^i_{i,b}\,. \tag{2.28}$$

Changing the inertial reference frame to a frame r located on the body's c. m. leads to the shortest form of the angular momentum; additionally, the coordinate system is changed to the body system in which the moment of inertia matrix is constant:

$$\underline{h}^b_{r,\mathbf{B}} = \underline{\underline{I}}^b_{\mathbf{B}}\underline{\omega}^b_{r,b} \tag{2.29}$$

Sometimes the moment of inertia of the body or a part of the body $\underline{\underline{I}}^b_{\mathbf{C}}$ is known relative to a frame c but not relative to the body frame b. Similar to equation 2.21 the radius vector in the body frame is split up into two parts. First, $\underline{r}^b_{b,c}$ is the vector that describes the displacement between both frames, and the second vector describes the point masses relative to this new frame c.

$$\underline{r}^b_{b,N} = \underline{r}^b_{b,c} + \underline{r}^b_{c,N} \tag{2.30}$$

Additionally, the sum of all point masses is the total mass:

$$\sum_N m_N = m_{\mathbf{C}} \tag{2.31}$$

Rewriting equation 2.27 with the vector triangle from equation 2.30 leads to

$$\begin{aligned}
\underline{\underline{I}}^b_{\mathbf{B}}\underline{\omega}^b_{i,b} &= \sum_N \left[m_N\left(\underline{r}^b_{b,c} + \underline{r}^b_{c,N}\right) \times \left(\underline{\omega}^b_{i,b} \times \left(\underline{r}^b_{b,c} + \underline{r}^b_{c,N}\right)\right)\right] \\
&= \underbrace{\sum_N \left[m_N \underline{r}^b_{c,N} \times \left(\underline{\omega}^b_{i,b} \times \underline{r}^b_{c,N}\right)\right]}_{\underline{\underline{I}}^b_{\mathbf{C}}\underline{\omega}^b_{i,b}} + \underbrace{\sum_N \left[m_N \underline{r}^b_{b,c} \times \left(\underline{\omega}^b_{i,b} \times \underline{r}^b_{b,c}\right)\right]}_{m_{\mathbf{C}}\underline{r}^b_{b,c}\times\left(\underline{\omega}^b_{i,b}\times\underline{r}^b_{b,c}\right)} \\
&\quad + \underbrace{\sum_N \left[m_N \underline{r}^b_{c,N}\right]}_{\overset{\text{c.m.}}{=}0} \times \left(\underline{\omega}^b_{i,b} \times \underline{r}^b_{b,c}\right) + \underline{r}^b_{b,c} \times \left(\underline{\omega}^b_{i,b} \times \underbrace{\sum_N \left[m_N \underline{r}^b_{c,N}\right]}_{\overset{\text{c.m.}}{=}0}\right),
\end{aligned} \tag{2.32}$$

with the abbreviation for the known moment of inertia $\underline{\underline{I}}^b_{\mathbf{C}}$ and with vanishing terms if the reference frame c is on the c. m. of the body part. This concludes the angular momentum of the body $\mathbf{C}$ with respect to the displaced body frame b:

$$\underline{\underline{I}}^b_{\mathbf{B}}\underline{\omega}^b_{i,b} = \underline{\underline{I}}^b_{\mathbf{C}}\underline{\omega}^b_{i,b} + m_{\mathbf{C}}\underline{r}^b_{b,c} \times \left(\underline{\omega}^b_{i,b} \times \underline{r}^b_{b,c}\right) \tag{2.33}$$

This equation includes the Huygens-Steiner *theorem* or the *parallel axis theorem* [1].

Dynamics

As for the translational dynamics in equation 2.2 the derivative of the angular momentum from equation 2.20 must also be carried out in an inertial frame for the rotational dynamics:

$$\begin{aligned}
\frac{\mathrm{d}_i \underline{h}^i_{i,\mathbf{B}}}{\mathrm{d}t} &= \frac{\mathrm{d}_i}{\mathrm{d}t} \sum_N \left[\underline{r}^i_{i,N} \times \underline{p}^i_{i,N} \right] \\
&= \sum_N \left[\frac{\mathrm{d}_i \underline{r}^i_{i,N}}{\mathrm{d}t} \times \underline{p}^i_{i,N} \right] + \sum_N \left[\underline{r}^i_{i,N} \times \frac{\mathrm{d}_i \underline{p}^i_{i,N}}{\mathrm{d}t} \right] \\
&= \sum_N \left[[m_N \underbrace{\frac{\mathrm{d}_i \underline{r}^i_{i,N}}{\mathrm{d}t}}_{=\underline{v}^i_{i,N}} \times \underline{v}^i_{i,N} \right] + \sum_N \left[\underline{r}^i_{i,N} \times \underbrace{m_N \frac{\mathrm{d}_i \underline{v}^i_{i,N}}{\mathrm{d}t}}_{=\underline{F}^i_N} \right] \\
&= \sum_N \left[\underline{r}^i_{i,N} \times \underline{F}^i_N \right] = \sum_N \underline{T}^i_{i,N}
\end{aligned} \tag{2.34}$$

This derivative shows that the inertial change in angular momentum is equal to the sum of all acting torques $\underline{T}^i_{i,N}$ on each point mass or the overall torque $\underline{T}^i_{i,b}$.

$$\frac{\mathrm{d}_i \underline{h}^i_{i,\mathbf{B}}}{\mathrm{d}t} = \sum_N \underline{T}^i_{i,N} = \underline{T}^i_{i,b} \tag{2.35}$$

The inertial derivation will now be transformed to the body frame to utilize the above mentioned fact that the moment of inertia tensor is constant in this frame. As an intermediate step the angular momentum from equation 2.35 is redefined for the reference frame r which is located at the c. m. of the body, but does not rotate relatively to the inertial frame. Still, the derivative has to be applied in the inertial frame. Applying the Euler-transformation, like equation 2.22, to the change of angular momentum of body $\mathbf{B}$ relative to frame r results in

$$\frac{\mathrm{d}_i \underline{h}^i_{r,\mathbf{B}}}{\mathrm{d}t} = \frac{\mathrm{d}_r \underline{h}^i_{r,\mathbf{B}}}{\mathrm{d}t} + \underline{\omega}^i_{i,r} \times \underline{h}^i_{r,\mathbf{B}} \,, \tag{2.36}$$

where $\underline{\omega}^i_{i,r}$ is the angular velocity of r with respect to i which is zero by definition. Now it is possible to transform the change of angular momentum to the body fixed frame by again applying the Euler-transformation:

$$\frac{\mathrm{d}_r \underline{h}^i_{r,\mathbf{B}}}{\mathrm{d}t} = \frac{\mathrm{d}_b \underline{h}^i_{r,\mathbf{B}}}{\mathrm{d}t} + \underline{\omega}^i_{r,b} \times \underline{h}^i_{r,\mathbf{B}} = \underline{T}^i_{r,b} \tag{2.37}$$

After changing the coordinate system to the body fixed and using equation 2.29, the reason

for changing the frames becomes evident.

$$\frac{\mathrm{d}_b\left(\underline{\underline{I}}^b_{\mathbf{B}}\underline{\omega}^b_{r,b}\right)}{\mathrm{d}t} + \underline{\omega}^b_{r,b} \times \left(\underline{\underline{I}}^b_{\mathbf{B}}\underline{\omega}^b_{r,b}\right) = \underline{T}^b_{r,b} \tag{2.38}$$

$$\underbrace{\frac{\mathrm{d}_b\underline{\underline{I}}^b_{\mathbf{B}}}{\mathrm{d}t}}_{=0}\underline{\omega}^b_{r,b} + \underline{\underline{I}}^b_{\mathbf{B}}\frac{\mathrm{d}_b\underline{\omega}^b_{r,b}}{\mathrm{d}t} + \underline{\omega}^b_{r,b} \times \left(\underline{\underline{I}}^b_{\mathbf{B}}\underline{\omega}^b_{r,b}\right) = \underline{T}^b_{r,b} \tag{2.39}$$

The first derivative is zero because the moment of inertia is constant in the body fixed frame.

So far the discussed body has not been defined in any way. The goal of the calculations of the rigid body's dynamics is to describe the rotation of a satellite in the orbit plane. The satellite in the following example consists of the satellite's structure and multiple reaction wheels to influence the attitude of the satellite itself. Reaction wheels work with spinning masses, which violates the assumption of a rigid body, but because it is safe to assume that the spinning masses are rotationally symmetric this makes no difference, except that the angular momentum of the wheels must be taken into account [5]. With these restrictions, the angular momentum of the satellite with multiple reaction wheels $\mathbf{W}_K$ is

$$\begin{aligned}\underline{h}^b_{r,\mathbf{B}} = \quad & \underline{\underline{I}}^b_{\mathbf{S}}\underline{\omega}^b_{r,b} + m_{\mathbf{S}}\underline{r}^b_{b,s} \times \left(\underline{\omega}^b_{r,b} \times \underline{r}^b_{b,s}\right) \\ & + \sum_K \left[\underline{\underline{I}}^b_{\mathbf{W}_K}\underline{\omega}^b_{r,w_K} + m_{\mathbf{W}_k}\underline{r}^b_{b,w_K} \times \left(\underline{\omega}^b_{r,w_K} \times \underline{r}^b_{b,w_K}\right)\right] ,\end{aligned} \tag{2.40}$$

with $\underline{\underline{I}}^b_{\mathbf{S}}$ and $\underline{\underline{I}}^b_{\mathbf{W}_K}$ being the moment of inertia tensors of the satellite's structure and the Kth wheel, respectively.

For application of the *parallel axis theorem* from equation 2.33, also given are their respective masses and their individual terms for the displacement from the combined c. m. $\underline{\omega}^b_{r,b}$ is the angular velocity of the satellite with respect to the reference frame and $\underline{\omega}^b_{r,w_K}$ is the spinning rate of a reaction wheel plus the satellite's rate:

$$\underline{\omega}^b_{r,w_K} = \underline{\omega}^b_{r,b} + \underline{\omega}^b_{b,w_K} \tag{2.41}$$

Equation 2.40 and its derivative with respect to time can be inserted into equation 2.37. The resulting first order differential equation can be simplified by assuming that the inertia tensors of the reaction wheels $\underline{\underline{I}}^b_{\mathbf{W}_K}$ are part of the inertia tensor of the whole satellite $\underline{\underline{I}}^b_{\mathbf{B}}$. This is a valid assumption because in most cases the inertia tensor of a real satellite is measured as a whole prior to the launch in a special facility.

The differential equation describing the motion of a satellite with reaction wheels for attitude control reads as follows:

$$\begin{aligned}\frac{\mathrm{d}_b\underline{\omega}^b_{r,b}}{\mathrm{d}t} = \ & \underline{\underline{I}}^{b\,-1}_{\mathbf{B}}\Bigg(\underline{T}^b_{r,b} - \underline{\omega}^b_{r,b}\times\left(\underline{\underline{I}}^b_{\mathbf{B}}\underline{\omega}^b_{r,b}\right) \\ & - \underline{\omega}^b_{r,b}\times\sum_K\left[\underline{\underline{I}}^b_{\mathbf{W}_K}\underline{\omega}^b_{b,w_K}\right] - \sum_K\left[\underline{\underline{I}}^b_{\mathbf{W}_K}\frac{\mathrm{d}_b\underline{\omega}^b_{b,w_K}}{\mathrm{d}t}\right]\Bigg)\end{aligned} \tag{2.42}$$

Any other torques acting on the satellite, e. g. disturbances, are described by the term $\underline{T}^b_{r,b}$.

Kinematic

For a complete description of the satellite's motion an attitude representation is necessary and quaternions will be used for this purpose. Especially in spacecraft simulation quaternions are useful because they describe attitudes in a singularity-free way.

The quaternion is an extension of the *complex plane* with three imaginary components i, j and k. The following definitions and notations for quaternion computation are taken from [6, 7]. A quaternion can be described in the form of summation

$$\mathbf{q} = q_4 + iq_1 + jq_2 + kq_3\,, \tag{2.43}$$

or in form of a vector:

$$\mathbf{q} = \begin{Bmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \end{Bmatrix} \tag{2.44}$$

A unit quaternion represents an attitude or a rotation. It describes a rotation around a specified axis $\underline{n}$, of unit length, by an angle of ε. The definition of such a rotation is

$$\mathbf{q} = \begin{Bmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \end{Bmatrix} = \begin{Bmatrix} \sin\left(\frac{\varepsilon}{2}\right) n_1 \\ \sin\left(\frac{\varepsilon}{2}\right) n_2 \\ \sin\left(\frac{\varepsilon}{2}\right) n_3 \\ \cos\left(\frac{\varepsilon}{2}\right) \end{Bmatrix}, \text{ with } |\underline{n}| = 1 \Rightarrow |\mathbf{q}| = 1\,. \tag{2.45}$$

If $\mathbf{q}'$ is the attitude of a satellite and $\mathbf{q}$ is a rotation which a satellite has to carry out, then the quaternion product of both is the resulting attitude $\mathbf{q}''$ after the maneuver:

$$\mathbf{q}'' = \mathbf{q}\mathbf{q}' \tag{2.46}$$

The opposing rotation is described by the *conjugated* quaternion $\overline{\mathbf{q}}$ where the signs of the first three elements are reversed, similar to complex numbers. The following is the rule for multiplying two quaternions:

$$\begin{Bmatrix} q''_1 \\ q''_2 \\ q''_3 \\ q''_4 \end{Bmatrix} = \begin{Bmatrix} q'_4 & q'_3 & -q'_2 & q'_1 \\ -q'_3 & q'_4 & q'_1 & q'_2 \\ q'_2 & -q'_1 & q'_4 & q'_3 \\ -q'_1 & -q'_2 & -q'_3 & q'_4 \end{Bmatrix} \begin{Bmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \end{Bmatrix} \tag{2.47}$$

One very important property of such a rotation is that no evaluation of a sine function is necessary, except for building the quaternion itself. For example, a repetitive task of carrying out the same rotation over and over again is computationally more efficient than most other attitude representations.

The calculation rule can be approximated for small rotation angles ε and then be used to build a linear differential equation for the attitude kinematics depending on the angular velocity $\underline{\omega}$[5]:

$$\frac{\mathrm{d}\mathbf{q}}{\mathrm{d}t} = \frac{1}{2}\begin{Bmatrix} 0 & \omega_3' & -\omega_2' & \omega_1' \\ -\omega_3' & 0 & \omega_1' & \omega_2' \\ \omega_2' & -\omega_1' & 0 & \omega_3' \\ -\omega_1' & -\omega_2' & -\omega_3' & 0 \end{Bmatrix}\mathbf{q} = \frac{1}{2}\underline{\underline{\Omega}}\mathbf{q} \tag{2.48}$$

When simulating the attitude dynamics with equation 2.42, equation 2.48 is used to calculate the change in attitude. These two equations suffice to simulate the motion of a spacecraft around its c. m. and with equation 2.3 a numerical simulation of all six degrees of freedom is possible.

Often rotations are needed in form of transformation matrices to change between different coordinate systems. If two coordinate systems a and b are rotated with respect to each other by a quaternion $\mathbf{q}_b^a$, then the transformation matrix of system a with respect to b is:

$$\underline{\underline{T}}_b^a(\mathbf{q}_b^a) = \begin{bmatrix} q_1 - q_2 - q_3 + q_4 & 2\left(q_1q_2 + q_3q_4\right) & 2\left(q_1q_3 - q_2q_4\right) \\ 2\left(q_1q_2 - q_3q_4\right) & -q_1 + q_2 - q_3 + q_4 & 2\left(q_2q_3 + q_1q_4\right) \\ 2\left(q_1q_3 + q_2q_4\right) & 2\left(q_2q_3 - q_1q_4\right) & -q_1 - q_2 + q_3 + q_4 \end{bmatrix} \tag{2.49}$$

The transformation matrix describes only the rotational part of a coordinate systems change and not the translational part.

2.2 Digital interfaces

This section discusses the most common digital interfaces used on the test platform to connect the on-board computer of the experimental satellite to the sensors and actuators. The most basic characteristic of these links is a serial transmission of the digital contents. This means the smallest units of digital information, i. e., a bits, are sent one after the other, and multiple bits are combined into bigger packets of information, successively sent over the same communication line. Afterwards, the information has to be decoded and reassembled by the communication partner. This requires a communication protocol used by all involved interfaces to guarantee a loss-free exchange of information.

2.2.1 Electrical and data characteristics of serial standards

EIA-232

The EIA-232 link is a very wide-spread serial data exchange norm from the *Electronic Industries Association*. Often it is called *RS-232* or just *serial port*, which of course is a very

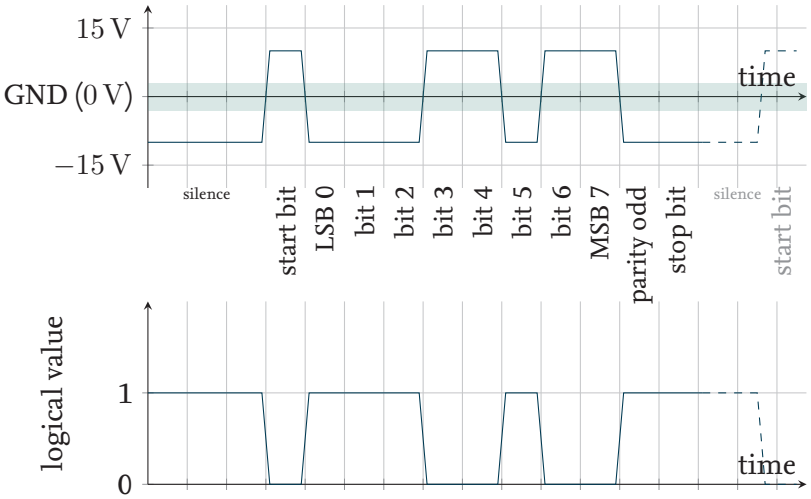


Figure 2.5: EIA-232 signals and data encoding

generic term. Fig. 2.5 shows the electrical encoding of multiple bits over time. It shows a very common way of interpreting the EIA-232 norm. The user data in this frame consists of eight bits forming one byte. It would also be possible to use fewer bits for the user data. The frame including the user data has multiple control bits, which are needed for the transmission and data integrity. To synchronize the receiving interface with the impending data a *start bit* is used. Following this the user data is transmitted from the least significant bit (LSB) to the most significant bit (MSB). This means the bits of a single byte are sent from the lowest (2^0) to the highest (2^7) valued bit of the binary stream. They can be followed by a parity bit to check for transmission errors. The *odd* parity bit is set to one if the user data contains an even amount of ones. Thus the total amount of ones in the frame is odd. Conversely, an *even* parity bit can also be used for the serial communication.

Another predefined property of an EIA-232 link is the time that has to pass before the next bit is allowed to be sent. The unit of measurement for this frequency is often "baud" which is the same as $1\,\mathrm{bit/s}$ At least in the EIA-232 and similar norms the baud rate includes all control bits, which significantly reduces the bandwidth of the communication channel. For this example eleven bits are needed to transmit one byte. Nowadays, typical baud rates are multiples of $19200\,\mathrm{bit/s}$, but others are possible as well. Fig. 2.6 shows the

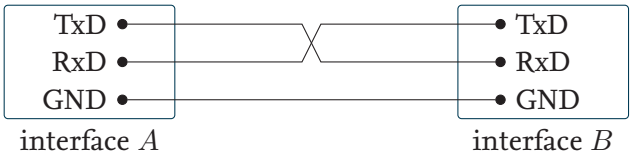


Figure 2.6: Typical wiring of an EIA-232 connection

standard three wire connection of two EIA-232 interfaces A and B. A lot more wires are

possible for setting up a connection but in most cases three suffice. The data transmission line is called *TxD*, whereas *RxD* is the receiving line. Interface A sends its data through TxD, which is connected to the receiving line of B. This is the reason why the wires in Fig. 2.6 are crossed.

The logical values are defined through the voltage between the ground GND and the transmission or the receiving line, respectively. The marked interval in Fig. 2.5 between $-3\,\mathrm{V}$ and $3\,\mathrm{V}$ is undefined, lower levels indicate a *one* and higher levels indicate a *zero*. In this norm the maximum allowed levels are $\pm 15\,\mathrm{V}$.

EIA-422

The EIA-422 norm has some similarities to the one in the previous section 2.2.1. It is also a serial transmission protocol and the control bits are identical to the ones used before. The most important difference between the two is shown in Fig. 2.7: The transmission of logical values happens via differential signals. The voltage to determine the logical value is

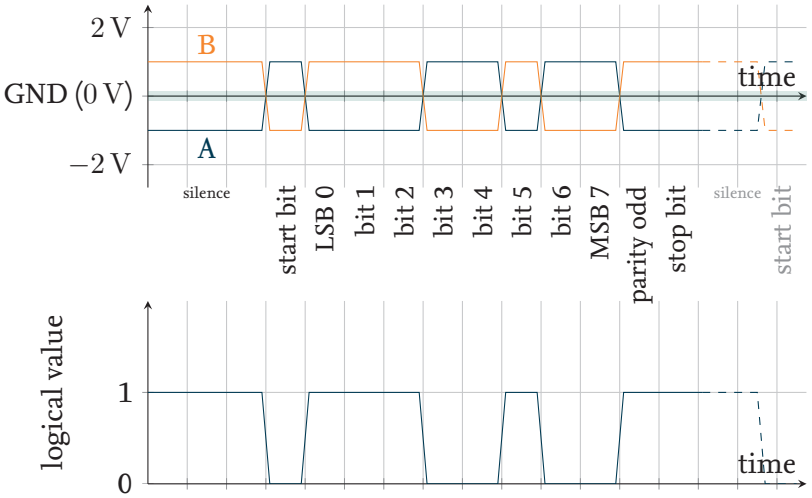


Figure 2.7: EIA-422 signals and data encoding

not measured between GND and a signal line, but between two different signal lines. The advantage of this is a better signal quality with less noise, enabling higher possible data rates, or a longer communication distance with lower voltage levels. The signal quality is better because disturbances are very likely to influence both signal wires in the same way such that there is no change in the differential signal.

The voltage levels must exceed $\pm 200\,\mathrm{mV}$ at the receiving interface to decode the signal correctly, compared to $\pm 3\,\mathrm{V}$ in the EIA-232 norm. The hardware can be more sensitive, which is made possible through the differential transmission. This norm makes it also easier in the definition of the upper voltage limit, Because most integrated circuits work with a $5\,\mathrm{V}$ power supply this norm simplifies the construction of interfaces, due to the lower upper voltage limit, as compared to the EIA-232.

The disadvantage is that more wires are necessary to connect two interfaces to each other as depicted in Fig. 2.8: Two wires for the transmission channel from interface A and two

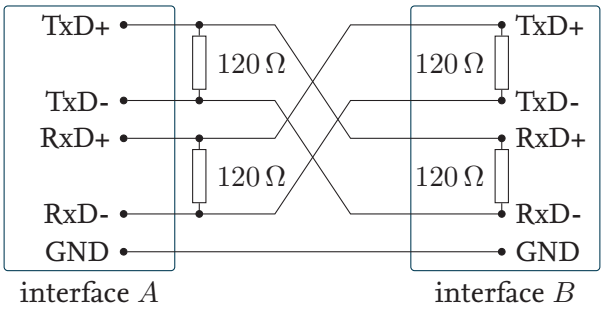


Figure 2.8: Typical wiring of an EIA-422 connection

for receiving the transmission from B. The ground connection is not mandatory but commonly used to prevent problems that could occur if the to communication partners lie on different ground potentials.

For longer communication distances it is recommended to use termination resistors at the end of each differential pair to prevent reflections, which would degrade the link quality. Theses resistors are sometimes located inside the interfaces themselves, but in other cases they have to be put in manually.

Such a configuration is capable of a transmission rate of several megabaud or $\mathrm{Mbit/s}$. Imaging sensors like cameras, star trackers or interferometers, communication links often require data rates in the magnitude of $\mathrm{Gbit/s}$, but in many cases an EIA-422 interface is sufficed for sensor and actuators while being inexpensive, reliable and having a low power consumption.

3 FACE – Facility for Attitude Control Experiments

This chapter describes the FACE in full detail. It is a laboratory where an attitude control system (ACS) of a small satellite can be demonstrated with real hardware. Of course the demonstration has some restrictions and cannot be fully as accurate as if the satellite were on an orbit in space. This test facility simulates the most important physical effect of the space environment, which is zero gravity, by reducing the gravitational impact on the simulated satellite through supporting it on an air bearing with its center of mass (c. m.) very close to its pivot point. This way the platform can rotate freely on the air bearing with almost no effects caused the gravitational acceleration. Additionally the Earth's magnetic field and the light of the Sun can be simulated with special equipment, which is also depicted in the fully assembled facility in Fig. 3.1.

Figure 3.1: Fully assembled test facility

An air bearing is used to reduce the friction to a minimum such that the experimental satellite can fulfill motions very close to a real satellite in space, but with restrictions in the total deflection in two of the three major axes.

This chapter is divided into multiple sections, each describing one individual component of the FACE. First, the components for the simulation of the space environment are

discussed; afterwards, the equipment for the ACS of an experimental satellite is described in detail.

3.1 Space environment simulator

For the simulation of the space environment, several different devices—whose number may increase in the future—are available and will be discussed in this section. The descriptions are based on the technical documentation [1, 2] and the experience gained during the installation and initial operations.

3.1.1 Magnetic field simulator

The magnetic field simulator, shown in Fig. 3.2, is a set of six coils or three Helmholtz coil pairs with a support assembly made of aluminum for magnetic neutrality. Each pair

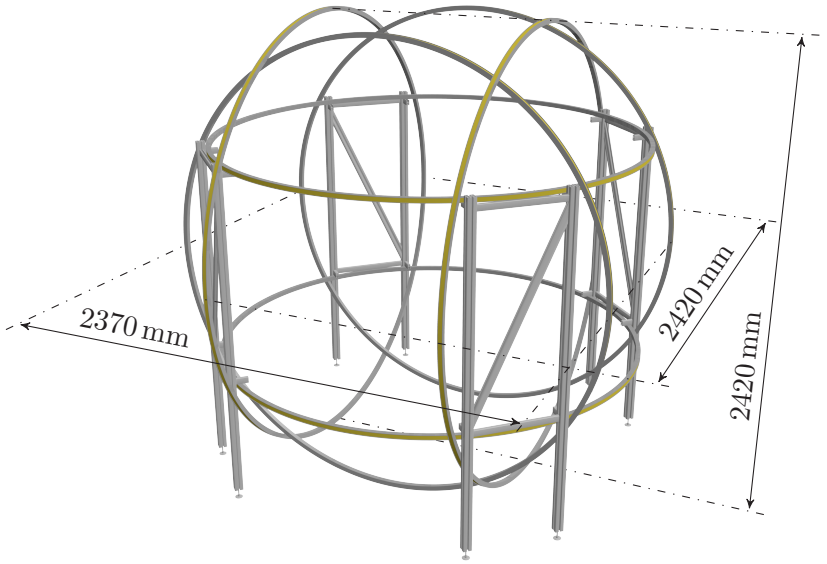


Figure 3.2: Magnetic field simulator

mainly influences one major axis of the magnetic field in the center of all coils. Each has a diameter of about $2.40\,\mathrm{m}$; the distance between the two coils of a pair is the coil radius itself. The strength of the resulting magnetic field is $\pm 210\,\mu\mathrm{T}$ per coil pair. Compared to the Earth's magnetic field, this is more than three times as high as the strength on ground level. But it has to be considered that for an accurate simulation a part of the magnetic capability is necessary to compensate for the local magnetic field.

This circular coil configuration was chosen for the quality of the resulting magnetic field, which is almost uniform, and by controlling the current in the coils a very precise simulation of the Earth magnetic field is possible. While circular coils were chosen for an accurate magnetic field [3], this choice complicates the construction of the support

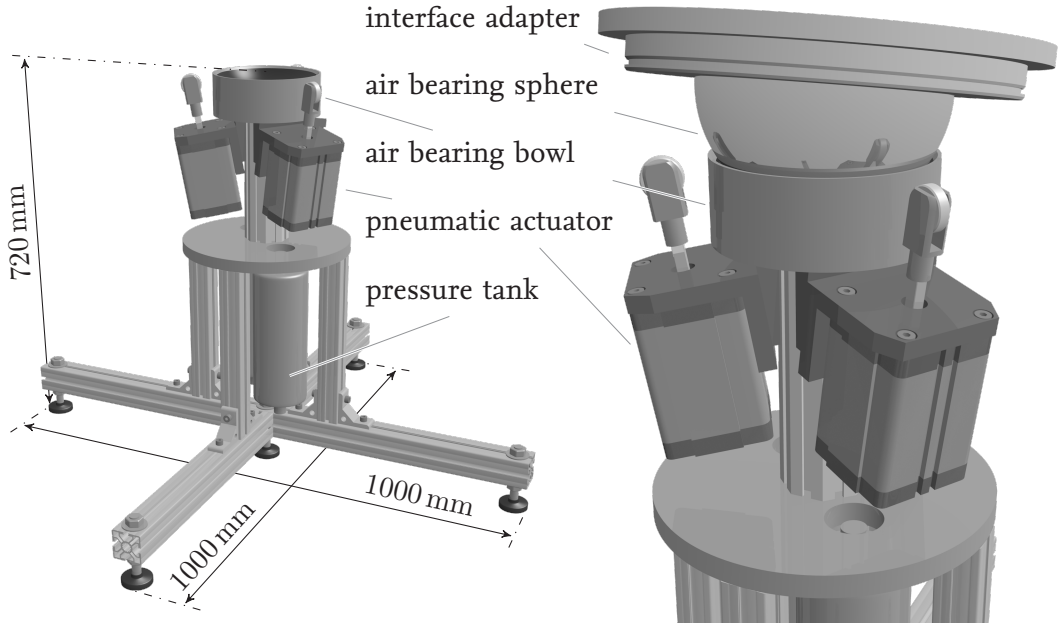
framework and thus the process to assemble the coils in the laboratory. The picture already hints this complication.

Each pair is connected in a series to a power supply that guarantees the same current in both coils. The three power supplies are also connected to each other via a data interface which allows them to be controlled by a computer. This link creates the possibility to control the magnetic field through simulation software that calculates the Earth magnetic field on the basis of an orbit propagation.

The controllable field is necessary for experiments with magnetic torquers as attitude actuators that are introduced in section 3.2.3, and to verify the tolerance of hardware against magnetic disturbances. Additionally, magnetometers can measure the simulated earth's magnetic field for a coarse attitude determination.

3.1.2 Support stator for air bearing

After setting up the magnetic coils, the support with the stator for the air bearing is lifted into the coil assembly, aligned at the center, and adjusted with a water level. The main component of the stator support is the bowl into which the counterpart of the bearing will be bedded (cf. Fig. 3.3(b)). Fixed on top of the air bearing (semi) sphere is an interface



(a) Support stator for air bearing

(b) Both parts of the air bearing

Figure 3.3: Support assembly on which the experimental satellite is loaded

adapter which later which will be attached to the assembly station (section 3.1.3).

As long as pressurized air flows between both parts of the bearing it can be used safely, but in case of a pressure drop, the sphere and the bowl will come into contact. In such a

situation the precisely manufactured bearing must not be allowed to move in any way to prevent a possible damage [1].

To guarantee safe usage, three pneumatic actuators are used to rotate the bearing and the assembly station back to a neutral position before the pressurized air can be shut of; these are visible in both parts of Fig. 3.3 in their retracted state. In case of an emergency, e. g., power outage or drop in pressure from the compressor, the breakdown mode is immediately activated and the pneumatic actuators are deployed [2]. Two pressure tanks are attached to the support to deliver energy needed in such an emergency.

Around the vertical axis the bearing can be rotated without limits, but both other axes have a maximal deflection angle of about $30°$, at which the interface adapter comes into contact with the air bearing bowl. Attaching the assembly station (introduced in the next section) reduces the deflection limits to $20°$ per axis because counter weights on the lower side of the assembly station might touch the structure of the support stator.

3.1.3 Assembly station

The assembly station in Fig. 3.4 is the platform onto which the ACS hardware can be mounted. The dimensions of the base plate are $800\,\mathrm{mm} \times 800\,\mathrm{mm}$ with a grid of bores

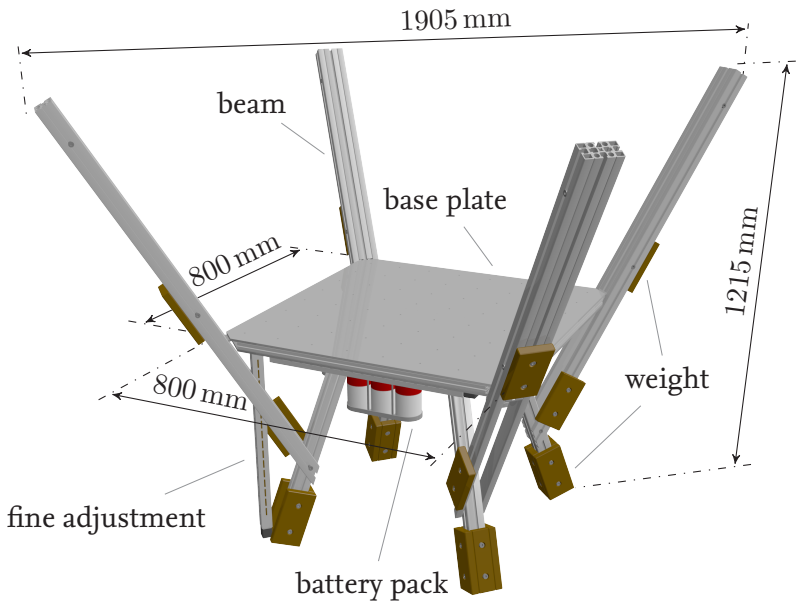


Figure 3.4: Assembly station for experimental satellite

to screw the attitude hardware tightly onto the top of the assembly station that has a total mass of more than $100\,\mathrm{kg}$. The eight beams at the corners of the platform hold brass weights, which are used to adjust the c. m. and also to influence the moment of inertia of the experimental satellite. The adjustment of the heavy brass weights is a very coarse method which will not suffice for accurate experiments. Therefore, for each of the body's

major axes a small lineally movable weight is attached. These small weights are fixed on a threaded rod which can be rotated by a stepper motor, so that a fine adjustment can be made contactlessly via a wireless connection. The picture only shows the mechanism for the axis perpendicular to the base plate; the other two are beneath it.

The friction in the air bearing should be as low as possible, hence it would not be a good idea to use a special construction to energize the platform from the support stator or elsewhere. The experimental satellite must be self-sufficient and therefore a dedicated power supply is installed. Four battery packs that can supply the hardware for several hours are attached symmetrically to the lower side of the base plate. For the duration of an experiment, the assembly station is completely autonomous and the only connection to the laboratory is a wireless local area network (WLAN) between the satellite and the control computer.

3.1.4 Sun simulator

The Sun has a major impact on most areas of an ACS. It can be used as a reference direction for attitude determination with a sun sensor. Solar panels have to be aligned for maximal electrical power generation, and sometimes parts of the spacecraft or a payload have to avoid direct contact with sunlight, for example, telescopes or radiators for thermal dissipation.

For these reasons, the sun can be simulated with a spotlight on a height-adjustable tripod and a framework for a mirror (cf. Fig. 3.5), to illuminate the parts of the experimental

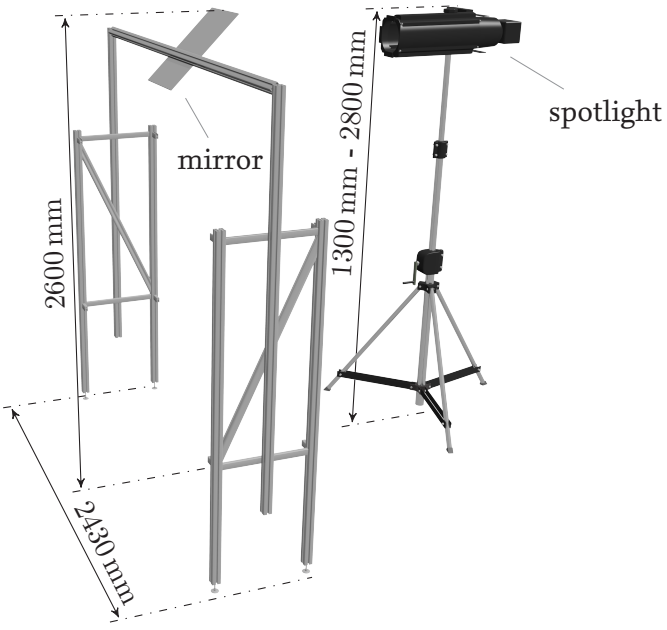


Figure 3.5: Sun simulator

satellite according to the specifications of the simulation experiment. If it is required

for an experiment, a direct illumination is also possible from the sides through the coil setup. In front of the spotlight there are an aperture and a lens to adjust the beam angle and focus. The intensity of the high pressure lamp can be controlled via digital multiplex (DMX), a standard protocol for stage lighting. Even tough precautions have to be taken to prevent damages to surfaces in the beam of the spot light [4], the light source is not able to simulate real sunlight for thermal verification tests. Another restriction for the simulation of the Sun is its fixed position during an experiment. This problem can be avoided by choosing an appropriate reference system for the air bearing table in which the Sun can be viewed as inertial.

3.2 Experimental satellite

The experimental satellite is the setup of all components that are used for the attitude control system (ACS): sensors, actuators and the on-board computer (OBC); this hardware mostly works the same way in the FACE as it would on a real satellite in space. This enables verification of the correct operation of the hardware itself, but also testing of control algorithms and the software architecture of the OBC.

3.2.1 On-board computer

Fig. 3.6 shows the OBC which will be used to control the experimental satellite. A normal

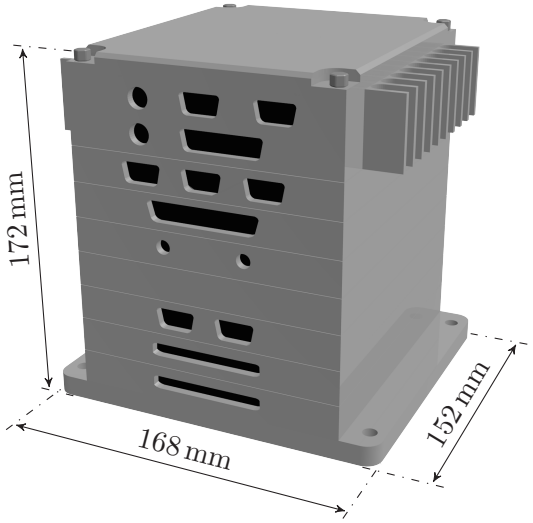


Figure 3.6: Industrial computer from RTD Embedded Technologies used as the OBC

non-space qualified computer for industrial applications has been chosen. There are two main reasons for this. First, it is much cheaper, and second, it is more powerful compared to space capable devices. This is important because the whole system has to be user friendly, which normally contradicts efficient usage of computation power. First designs

of ACS algorithms are usually not very well optimized, and debugging and the necessity of data exchange interfaces to a control computer in the laboratory take a lot of processing power. This anticipated overhead in the simulation environment in the FACE does not reflect the actual necessary computation power for the flight hardware. This means that in the end the design process results in a much more efficient algorithm for which a less powerful space qualified processor will most likely suffice.

Another advantage of the choice for industrial hardware is its expandability with interface boards, e. g., ethernet, serial ports [5], etc., which can be added without deeper knowledge of hardware design. The OBC consists of multiple modules stacked on top of each other. The cpu module [6] has the standard connectors for a monitor, keyboard, mouse, and two serial ports. For all other connections, separate modules are used, e. g., ethernet, WLAN, storage, and power module [7].

One module is only for serial connections that comply with the standard norms EIA-232, EIA-422 and EIA-485. The latter two norms are very similar, except that the EIA-485 standard supports multiple devices on one serial bus. Most of the sensors and actuators will be connected to this module, but a lot of different interfaces are available for future applications.

The computer runs the real-time operating system (RTOS) *QNX* that is mostly POSIX conform, which means it complies with a standard for application programming interfaces (APIs) that includes file and device handling, networking, etc. First of all, this has the advantage for UNIX and Linux users that they can work in a familiar environment, because the standard originated from UNIX-like operating systems (OSs).

The choice of QNX in favor of a Linux-based system is its real-time capability, which was a design goal for QNX contrary to the Linux real-time extensions which were developed on top of the existing operating system. An OS is called real-time capable if it can guarantee that it will react to a situation within a predefined time range. If the requirements are very vaguely specified, almost every OS can be called a (soft) RTOS.

However, for special applications very precise timing requirements have to be met. Such requirements may be how fast the system has to react to an external trigger, e. g., a sensor has finished a measurement and sends a signal to the computer to process this new data, or a local task on the OBC has to be executed every $\mathrm{X\,ms}$. For complying with theses requirements a (hard) RTOS is necessary.

If multiple programs are simultaneously working on this computer, only one is really running on the processor at any given time. The scheduling is a very important task of the OS. It takes care that important processes get enough computation time to ensure that real-time operations meet their specification, but also that other programs and the OS itself obtain cpu time. The QNX scheduler organizes all running programs that need processing time in a queue, to ensure an efficient usage of the computers processing power.

3.2.2 Reaction wheels

A very common device for influencing the attitude of a satellite is a reaction wheel. An electric motor controls a spinning mass and consequently the angular momentum of the

wheel. Accelerating the flywheel causes a counter reaction of the reaction wheel's casing mounted on the satellite. This changes the angular momentum of the remaining satellite. The accumulated angular momentum of the satellite and the spinning mass stays constant, but because the reaction wheel accelerates in one direction the whole satellite accelerates in the opposite direction. Equation 2.42 models this behavior when at least three reaction wheels perpendicular to each other are used to control the attitude of the satellite.

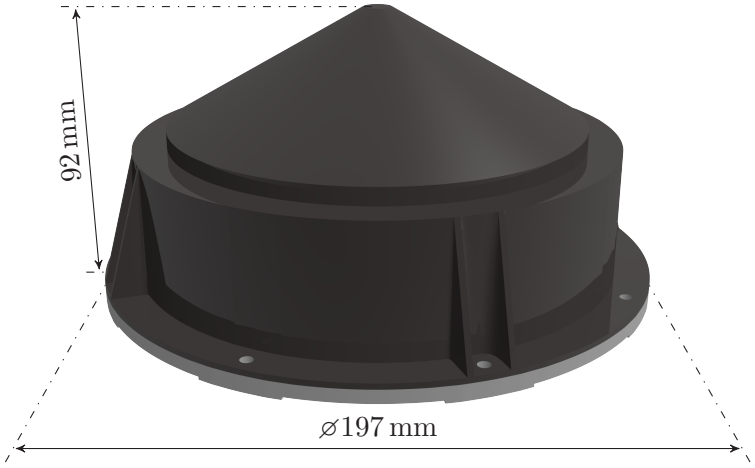


Figure 3.7: Reaction wheel RW 250 developed by Astro- und Feinwerktechnik Adlershof

Currently three *RW 250* (cf. Fig. 3.7) built by *Astro- und Feinwerktechnik Adlershof GmbH* are available for the experimental satellite. The orthogonal setup of three devices is depicted in Fig. 3.8 with their adapters for the attachment on top of the assembly station.

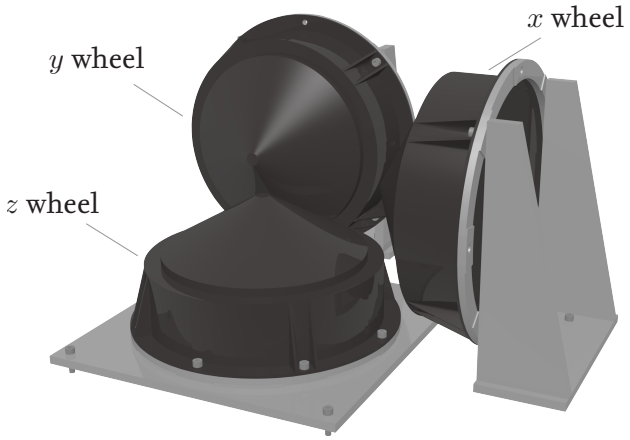


Figure 3.8: Orthogonal reaction wheel setup

In the near future a fourth wheel will be available for a redundant ACS that is tolerant

towards the failure of one reaction wheel. In that case, the reaction wheels will be aligned with the surface normals of a tetragon. In this arrangement the reaction wheels would not be perpendicular to each other and each wheel could influence, at least partly, the spin axis of another wheel.

Each of the devices has a mass of about $2.7\,\mathrm{kg}$ with a maximal commendable torque of $0.1\,\mathrm{Nm}$ which can be controlled at a maximal frequency of $2\,\mathrm{Hz}$. For the flywheel a moment of inertia of $5.5 \cdot 10^{-3}\,\mathrm{kgm}^2$ is given by the manual [8]. The range of the maximal rotation speed is $\pm 7000\,\mathrm{rpm}$, and by equation 2.35 the range of the angular momentum is $\pm 4\,\mathrm{Nms}$. By interchanging the angular momentum between wheel and satellite an unlimited amount of maneuvers would be possible, because the angular momentum is preserved. However if, say, the ACS has to hold a specific attitude under the influence of disturbances which are not regularly distributed, the wheel must be accelerated until it eventually reaches its maximum speed. At this point the wheel is saturated and another actuator is needed to reduce the angular rate of the wheel. Such an actuator must interchange angular momentum with the environment, e. g., a magnetic torquer or a propulsion system, and not an actuator based on the principal of shifting the angular momentum within the satellite.

3.2.3 Magnetic torquer

Even though there is currently no control hardware for the magnetic torquers in Fig. 3.9, the magnetic coils are mounted on top of the assembly station. As soon as the hardware is

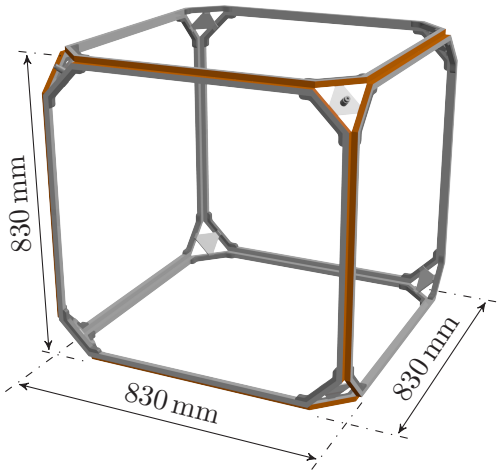


Figure 3.9: Assembly of paired magnetic torquers

available, it will be easier to recalibrate the table when the heavy coils are already attached and the c. m. and the pivot point are correctly adjusted.

Theses coils are the second type of actuator used for the experimental satellite. They can interact with the magnetic field simulator (sec. 3.1.1) in the FACE or with the Earth's magnetic field when used on a real satellite.

Like the HELMHOLTZ coils of the magnetic field simulator, three pairs of coils are used to generate a magnetic dipole, which is controllable in all three dimensions. This magnetic dipole moment $\underline{D}^b_{\text{coil}}$ and the Earth magnetic field $\underline{B}^b_{\text{Earth}}$, both with respect to the body frame, result in a torque perpendicular to both vectors:

$$\underline{T}^b_{i,b\text{dipol}} = \underline{D}^b_{\text{coil}} \times \underline{B}^b_{\text{Earth}} \,. \tag{3.1}$$

The magnetic dipole moment of a coil of n windings which encloses an area A with an electric current i is

$$\underline{D}^b_{\text{coil}} = niA\underline{e}^b \,, \tag{3.2}$$

where $\underline{e}^b$ defines the surface normal of A. On a $500\,\text{km}$ orbit, the strength of the magnetic field $\underline{B}^b_{\text{Earth}}$ is about $40\,\mu\text{T}$, depending on the location [2]. The coil pairs are dimensioned for $6\,\text{Am}^2$, which results in a maximal torque of

$$T^b_{i,b\text{dipol}} = 6\,\text{Am}^2 \cdot 40\,\mu\text{T} \approx 0.24\,\text{mNm} \,. \tag{3.3}$$

This torque will not be used directly to control the attitude of a satellite when other actuators are on-board, although this would be possible if no rotation around the magnetic vector is necessary. Equation 3.1 shows that it is not possible to generate a torque solely around the magnetic field vector. Additionally, the torque is very small and the accuracy is not adequate for high precision attitude control. But the torque is very valuable for the desaturation of reaction wheels to reduce the power consumption of high spinning flywheels or reducing the angular rate, if it is near to the design limits. In this case the magnetic torque is used to compensate the torque from a slowly decelerating flywheel.

3.2.4 Fiber optic gyroscope

A fiber optic gyroscope is a sensor for measuring angular rates. The term *gyroscope* is used despite the fact that there is no mechanical spinning mass. Before optic gyroscopes could be used to measure angular rates, because they were too expensive or simply not technically feasible, high spinning gyroscopes were used, and still are to the date. Therefore, a sensor for measuring angular rates is generally called *gyro*.

Three gyros from the company *LITEF GmbH* are available for the ACS on the air bearing table. This introduction of the µFORS-6U optic gyro (cf. Fig. 3.10) is based on the official documentation [9], which will also be used for the implementation of the software interface. The three gyros are attached to the bracket in Fig. 3.11 to align the sensitive axes of the devices with the body fixed frame of the assembly station.

The functionality of the used fiber optic gyroscopes are based on the *Sagnac* effect. The principle of this effect is depicted in Fig. 3.12 with its four main components.

The beam of a light source is split in two and both beams are passed through a coil of optical fibers in opposite directions. The length of such a fiber can be up to several kilometers coiled up in a small casing like Fig. 3.10. After emerging on the other side of the fiber, both beams are again merged with the beam splitter which leads them to a

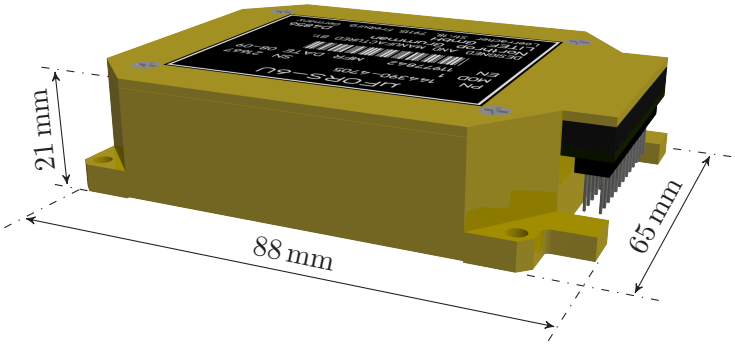


Figure 3.10: Rate gyro µFORS-6U developed by LITEF

detector. When the gyro is resting the traveling distance of both light rays stays constant over time, and the velocity is of course the speed of light. By turning the gyro around the axis perpendicular to the coil plane, one beam of light has to travel farther than the other. This leads to a phase shift between both merged rays at the detector, which measures the difference. The number of windings in the coil is proportional to the phase shift and therefore the sensitivity of the detector can be increased by using an appropriate amount of windings. Optic gyros can measure a broad spectrum of angular rates, from fractions of an arc second to thousands of degrees per second.

A typical characteristic of an optic gyro is its *random walk*. When the measurements of the detector are integrated over time, giving the total angle the gyro is rotated, the random walk describes a mean statistical error to the actual value. Because of noise and small measurement errors in the detector, the random walk is different from zero.

It can be used to calculate the noise on the sensor's signal. The standard deviation σ of the noisy signal depends on the random walk value η_{gyro} provided by the manufacturer of

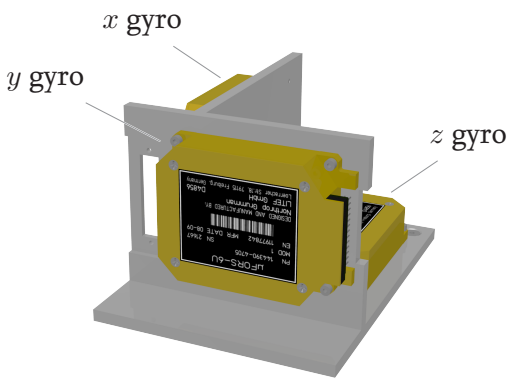


Figure 3.11: Orthogonal setup for the gyroscopes

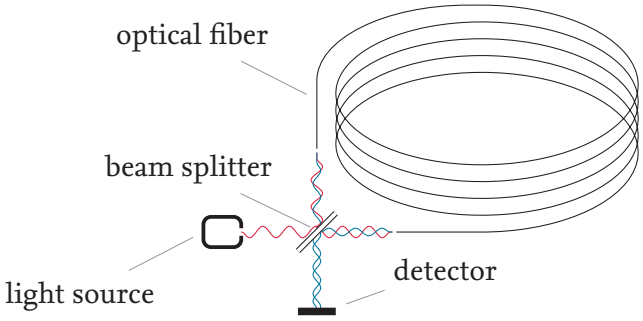


Figure 3.12: Principle of a fiber optic gyroscope based on the sagnac effect

the device, and the sampling frequency, which can be up to $1\,\mathrm{kHz}$.

$$\sigma_{\mathrm{gyro}} = \sqrt{\eta_{\mathrm{gyro}}^2 f_{\mathrm{sampling}}} \tag{3.4}$$

The user manual states for the µFORS-6U an η_{gyro} of $0.15°/\sqrt{\mathrm{h}}$. Later on, the standard deviation can be used to filter the signal to gain more trustworthy results. Other negative influences on the signal quality of theses gyros are the temperature and magnetic fields, but these are very small or, in the case of the temperature, are automatically compensated by the devices and will not be considered here.

3.2.5 Magnetometer

A magnetometer is a measuring instrument used to determine the magnetic field strength and its orientation. On board of a satellite it is used to chart the magnetic field of the Earth or to coarsely estimate the attitude in a known environment. Additionally, magnetic torquers can be monitored and checked for their correct operation. Two magnetometers are available for the experimental satellite; they differ in their basic measuring method. To isolate the magnetometers from the electromagnetic disturbances of the other attitude hardware, they are attached to the post in Fig. 3.13. The height of the devices relative to the assembly station is adjustable, this way they can be aligned with the center of the Helmholtz coils from the magnetic field simulator where the generated field is mostly homogeneous.

AMR magnetometer

The first magnetometer is manufactured by *ZARM Technik* [10] and uses the anisotropic magnetoresistance (AMR) effect to measure the magnetic field vector. The very compact device is depicted in Fig. 3.14. The AMR effect is a property of materials that change their electrical resistance under the influence of an external magnetic field. A thin, ferromagnetic metal sheet that is flowed by a current reaches its maximal resistance when the external magnetic field is parallel to the current and its minimal resistance when they are perpendicular to each other. To determine the three-dimensional magnetic field vector,

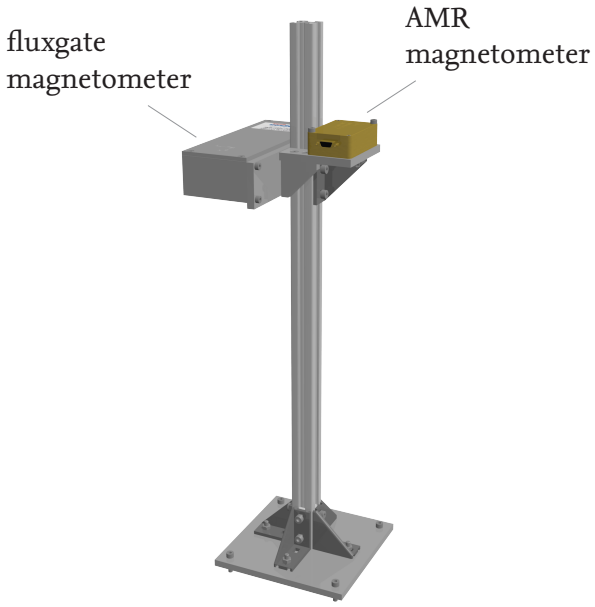


Figure 3.13: Post for the magnetometers

more than three metal strips are needed because the electrical resistance yields only directional information, not orientational. The measuring range of the AMR magnetometer is $\pm 250\,\mu\mathrm{T}$ with an accuracy of better of than $\pm 1\,\%$ for a measuring frequency of 6 Hz.

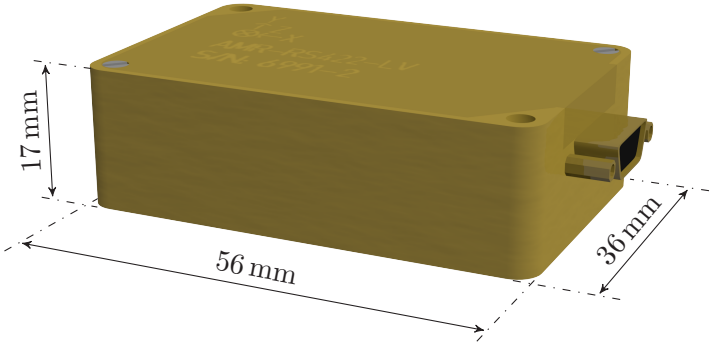


Figure 3.14: AMR magnetometer developed by ZARM Technik

Fluxgate magnetometer

A second physical principle that can be used for measuring a magnetic field is utilized by the *fluxgate* magnetometer developed by *Magson*. Six small magnetic coils are attached to the casing in Fig. 3.15, similar to a Helmholtz setup.

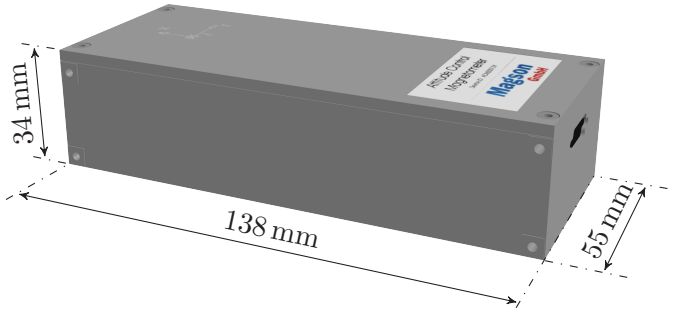


Figure 3.15: Fluxgate magnetometer developed by Magson

At the center of theses coils there are two soft-magnetic ringcores that are wrapped with a wire to drive the cores into saturation by using an alternating excitation current [3]. The resulting magnetic field induces a voltage in the six coils that is constantly measured. An external magnetic field distorts these received signals and could be used to calculate the magnetic field vector. For a more precise measurement, however, three additional coils are used to compensate the external magnetic field around the ringcores in all three spatial directions. After the external field is canceled out inside the sensor, the information about the power consumption of the compensating coils and the measurements of the six receiver coils are used to calculate the external field strength and orientation. About 50 measurements per second are possible for a measuring range of $\pm 180\,\mu\mathrm{T}$ and an accuracy of $\pm 0.25\,\%$. Especially the high sampling frequency is an advantage towards the AMR magnetometer, which on the other hand is smaller and has a lower power consumption.

3.2.6 Inertial measurement unit

The inertial measurement unit (IMU) *iIMU-FSAS* is a combination of optic rate gyros and accelerometers for all three spatial directions in one casing (cf. Fig. 3.16). The device

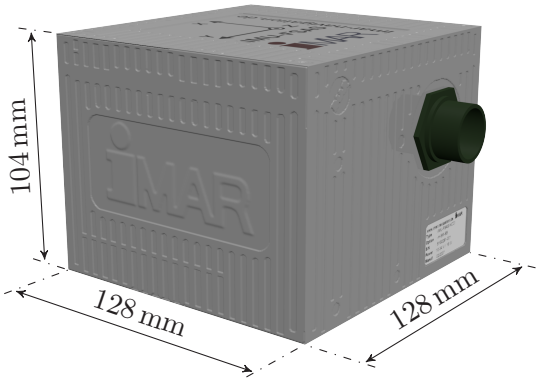


Figure 3.16: IMU developed by IMAR

uses two serial interfaces for the communication with an OBC. One is a standard EIA-422

interface that offers an interactive console to program the IMU. This console is used to change the sampling frequency of the sensors and to check the housekeeping data of the device [11].

The second serial interface is used to send the sensor values of the six sensors with the chosen frequency. It is a high data rate interface with a bandwidth of 2 Mbit for a very little delay when sending the sensor values over the serial line. Initial tests of the unit showed a correct operation, but it was not possible to receive the sensor values with the OBC. A special hardware interface is necessary to decode the signals which was not available at the time, but the link was observed with an oscilloscope that showed the proper functioning.

4 Development and implementation of the on-board communication

4.1 On-board communication layout

The main component of the on-board communication is the OBC. It is used to collect all sensor data, command the actuators and receive control instructions from an external source. Currently three communication standards are used for this purpose, namely EIA-422, EIA-485, and common WLAN. Fig. 4.1 shows the components with their corresponding link types.

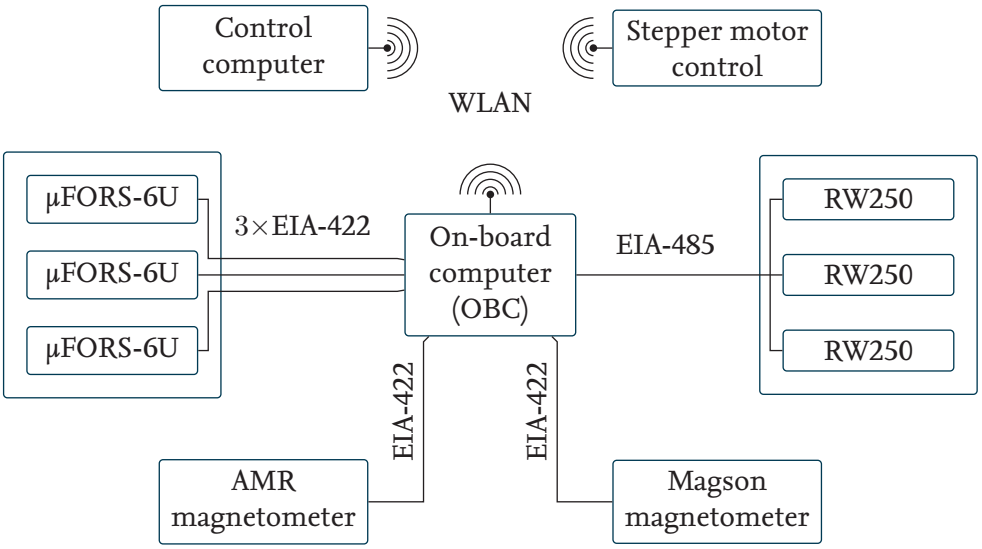


Figure 4.1: On-board communication layout of the experimental satellite

The three fiber optic gyros (µFORS-6U) are each connected to the on-board computer via an EIA-422 interface, while all the reaction wheels share a single EIA-485 bus. Both magnetometers have a standard differential serial interface (EIA-422). The OBC has a WLAN interface card for a wireless communication with the FACE control computer. This interface is also used to control the fine adjustment mechanism.

The control component for the stepper motors, which moves the weights along the major axes, is a commercial circuit board with an EIA-232 interface [12]. Because the fine adjustment of the c. m. has to be done contactless, a converter from EIA-232 to WLAN is mounted to the lower side of the assembly station. This way, the weights can be con-

trolled from a remote computer inside the laboratory. For the automation of this process the same link is used to control the steppers from the OBC for which the WLAN would not be necessary (the usage of the provided serial interface would be more practical). Nevertheless, the WLAN is used according to the design of the on-board infrastructure, hence no modifications are necessary.

The aim during the implementation of all protocols for the above devices is creating a user-friendly interface within the development environment of MATLAB™/Simulink™ with its ability to directly build real-time applications for the OBC with MATLAB Real-Time Workshop™ (RTW). Simulink™ is a graphical programming language for time-based simulations, enabling problem simulation and solving on a very high level. For instance, control problems look similar to graphical control diagrams as known from the literature, and solving of the corresponding differential equations happens in the background. Variable assignment, initialization and memory allocation happens automatically. This approach helps the user to concentrate on control problems, like identifying controller parameters and analyzing the results, and relieve them of the task to deal with the simulation itself.

The implementation of the attitude control hardware has the same intent. For later usage, the devices will have a graphical representation in Simulink™ with an interface that only outputs sensor data or accepts control commands. The internal structure of memory allocation, encoding and decoding of the communication packets, etc., will be transparent to the user.

4.2 Reaction wheel RW 250

The reaction wheels RW 250 are connected to the OBC via the serial standard EIA-485 that is capable of multiple clients on one data bus. The OBC acts as the host, and the wheels only talk in reaction to direct communication from the host. This way, all three (and later all four) wheels can share the same interface on the computer. Furthermore, only one data wire pair is used for sending and receiving, making the correct sequence of the communication essential. The host sends a packet and then waits for the response of the addressed wheel before sending the next request is sent.

4.2.1 Protocol

Two different protocols are described in [8] for the communication with the wheels: The *binary protocol*, to which none of the devices responded, and the *ascii protocol*, which is less efficient but could be implemented successfully. Tab. 4.1 shows the basic layout a packet sent between two communication partners.

Table 4.1: RW 250 packet layout and corresponding field sizes in byte

STX	addressee	addresser	command	data	checksum	EOX
1 B	1 B	1 B	1 B	0 B ... n B	2 B	1 B

Every packet begin with a start byte STX and is terminated by an end byte EOX. EOX is defined as the value 0x0d (hexadecimal notation). The start byte is 0x23 if the host sends a command and 0x32 if the packet is sent by a client. Enclosing the packet between these bytes happens for easier recognition of a packet, particularly the end of the packet, whose length can vary.

Since no communication is allowed between two wheels, one of the *addresser* or *addressee* fields is always 0x40, identifying the host, while the other field is the predefined and unique identification number of a wheel.

Nine different commands are defined by the protocol. Four commands control the flywheel's angular momentum in different operation modes.

0x6a (ω_{simple}) The reaction wheel accelerates the flywheel to the desired angular velocity using a proportional (P) controller and guaranteeing a steady state lower than the intended velocity.

0x65 (ω_{strat}) The reaction wheel accelerates the flywheel to the desired angular velocity with a proportional-integral (PI) controller.

0x63 ($\dot{\omega}_{\text{coarse}}$) The reaction wheel accelerates the flywheel with the desired angular acceleration by a PI controller.

0x69 ($\dot{\omega}_{\text{adaptive}}$) The reaction wheel accelerates the flywheel with the desired angular acceleration by an adaptive controller.

An attitude control maneuver generally uses the latter two commands, because the acceleration of the flywheel multiplied by its known moment of inertia equals the attitude control torque, which is calculated by the ACS. The command byte and the control value sent in the *data* field form the packet for the EIA-485 communication.

The next three commands have no direct influence on the angular momentum of the reaction wheel:

0x6c (set power mode) Chooses the upper limit of the overall power consumption of the device. The command values 0, 1, and 2 represent the limits of $50\,\mathrm{W}$, $100\,\mathrm{W}$, and $150\,\mathrm{W}$, respectively.

0x66 (clear control deviation) Sets an internal control deviation to zero.

0x72 (restart) In case of a malfunction the device can be restarted through this command.

While the previous commands are just acknowledged by the wheel through sending back a slightly modified command byte, ensuring that the command from the host was successfully transmitted, the final two commands request actual data to be sent back from the wheels.

0x74 (send housekeeping) The housekeeping data packet includes: the voltage and current of the electric motor circuit, motor and electronic circuit board temperature and the measured angular velocity of the flywheel;

0x6f (send data) This response data packet includes: the demanded control value (angular velocity/acceleration), the control deviation, which can be cleared with the 0x66 command, maximal commendable angular acceleration and the predicted angular velocity of the flywheel $125\,\mathrm{ms}$ after the packet is sent.

Before the MATLAB™/Simulink™ implementation of this protocol is explained in detail, a few important functions are required by this protocol.

4.2.2 Special functions

Encoding and decoding of numerical integer values

The implemented protocol is called ASCII because the values transmitted over the serial interface (angular velocity/acceleration, power mode, temperature) are converted into American Standard Code for Information Interchange (ASCII) characters. The table in Appendix D maps a hexadecimal digit to its ASCII representation. The integer values in this protocol include multiple hexadecimal digits that have to be encoded to ASCII one at a time. This encoding must be done for every command value to be sent to a reaction wheel and for every received byte the data must be decoded in the reversed order.

Checksum

Digital data transmission is always sensitive towards errors in the transmission. Data that is corrupted on the way from sender to receiver can be dangerous. It is not possible to completely prevent this corruption, but it is important to detect it. For this purpose redundant data is used. The simplest way would be to send every piece of data twice and look for discrepancies between both versions. If an error is detected, the data can be requested again until no error is found. Of course there is a possibility that the same error occurs during the transmission of both versions, but the probability of this happening is very small.

The protocol for the RW 250 implements a different checksum algorithm that adds all bytes from a packet (cf. Tab. 4.1) together, except for the two checksum bytes and the end byte ETX. The remainder when dividing this value by 256 results in an integer between 0 and 255 which can be stored in one byte. Using the algorithm from the section above, this $8\,\mathrm{bit}$ integer is encoded to its two bytes ASCII representation.

When a packet is received from a reaction wheel, then the calculated checksum is compared to the checksum at the end of the packet, and if there is a difference, an error occurred during the transmission and the corrupted data is discarded. A packet sent to the wheels must also include a checksum stored at the end of the packet. When the wheel receives this packet, it will conduct the identical error check.

4.2.3 RW 250 software interface

The reaction wheels are connected to the EIA-485 bus from Fig. 4.1. The serial port and the list of identification numbers of each individual reaction wheel are passed to the software interface. The software routine is called periodically from a timer in the Simulink™

framework, and in each run the software sends a packet or waits for an answer from one of the wheels. The flow chart for this process is depicted in Fig. 4.2 and shows the most important steps that have to be handled by the software repeatedly. The routine always

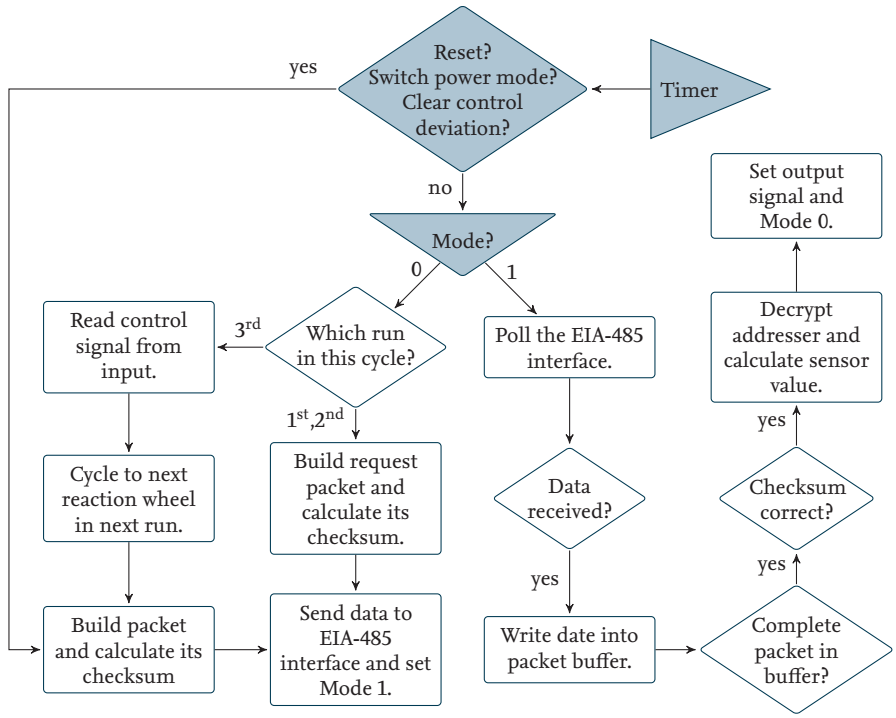


Figure 4.2: Flow chart of the RW 250 software interface

starts at the timer input but can end on multiple locations in the chart. Sometimes only one branch exists after a condition, meaning if the condition is not met, the routine exits. A typical example of such a situation is when a command packet was sent in the last run but no acknowledgment packet has been received since then. At this point the function stops and eventually, some runs later, a packet is received and the condition is met.

The function works with global variables to control the overall process. The process is driven by a wheel cycle that shifts through all registered reaction wheels and a second cycle that controls whether a command signal should be sent, the housekeeping should be requested, or, if an expected acknowledgment packet was not yet received, the routine should wait and do nothing.

The first step after entering the function is to check for special requests that should be handled immediately. This could be the signal to reset the micro controller of a reaction wheel, change the power limitations, or clear the control deviation.

Otherwise, in the standard case it is checked whether a new packet should be built and sent (Mode 0) or if a response packet has not been received yet (Mode 1). The latter case prohibits the usage of the single communication line of the EIA-485 bus for any other communication. In Mode 0 the next packet to be sent is chosen. The order in which

the packets are chosen is as follows. In the first and second run the two housekeeping data sets are requested, in the third run an actuator command to control the flywheel is sent. If it is the first run in the cycle, the packet is build according to Tab. 4.1 with the command 0x74; In the second run the other data set is requested with the command 0x6f. In both cases, the checksum of the packet is calculated and appended to the packet itself; It will be used by the reaction wheel to verify the integrity of the received command. The prepared packet is now ready and is sent to the serial interface. Because it is not allowed for the OBC to use the serial connection until the acknowledgment packet is received, the Mode is changed to 1 and the routine exits, until some time later it is again triggered by Simulink™.

If it is the third run in this cycle, a command packet is set up according to the user-chosen control type (angular velocity/acceleration) and command value (desired velocity or acceleration) encoded as described above. After the three runs, the wheel cycle value is changed to serve the next reaction wheel the next time it is possible to send a packet.

Each time the OBC sends a command it has to wait for the answer. In Mode 1 the serial interface is polled for any received data, and if any data is available it is stored in a buffer for further processing. The next step is to search the buffer for a complete packet, i. e. the STX byte, followed by some other bytes, followed by the packet indicator EOX. It is possible that just a part of the packet is received between two runs of the routine, and no ending byte is in the buffer. In this case, the function stops and when the rest of the packet is eventually received in a following run, it is written to the end of the buffer.

When a complete packet is found in the buffer, the first step is to verify the checksum. If this check fails the packet is discarded and the function is aborted without further consequences. This is acceptable because it does not happen very often, and the command will just be sent again the next time it is the particularly reaction wheel's turn. Usually, the test is passed and the packet is decoded in the case housekeeping data was sent. The Mode is changed back to 0, because the communication line is now available for the next command.

On some occasions no acknowledgment packet is sent back from a reaction wheel because some error occurred or the micro controller was reset. For this reason a function was implemented that is commonly called *watchdog timer*. When a packet is sent to the serial port, a stop watch is started. If this stop watch runs longer than a predefined time threshold the Mode it automatically reset to 0. This way the routine does not get stuck in the listening mode in case an anticipated packet does not appear.

The frequency for triggering the timer in Fig. 4.2 is subject to the following two boundary conditions. First, each wheel may be given a new control value at most twice a second (this is being ensured by the routine itself). Second no more than five seconds may pass between two control values being sent; otherwise the wheel switches to a fail safe mode to prevent damage to the wheel in case of a malfunction in the OBC. A frequency of $100\,\mathrm{Hz}$ suffices for three reaction wheels on a serial bus; with a fourth wheel a slightly higher value will be necessary. In the present setup the wheels are sent control data twice a second, and the housekeeping data is requested approximately ten times a second, which seems to be slightly higher than the frequency in which the measured angular rate of the fly wheel is updated on the device.

The source code for the software interface is written in C and uses a template for a *level-2 S-function* that works directly with MATLAB™ Simulink™. The source code can be found in appendix C.1 with a picture of the high level interface for Simulink™ in Fig. C.1. Most of the housekeeping data is not available at this level as it is not of interest to the user in the normal case. They only need access to the angular rate and acceleration, from which the total angular momentum (cf. equation 2.29) and torque (cf. equation 2.35) can be deduced.

4.3 Fiber optic gyroscope µFORS-6U

The optic gyroscopes manufactured by LITEF are connected to the OBC each with a separate EIA-422 interfaces. Thus, this means the software interface only communicates with one device and no identification numbers are needed. All three gyros use the same software of which three instances are running on the computer, each accessing a different serial port.

An extra hardware device is used to program the gyros to different modes with special properties. For example, the frequency of the communication link can be changed, or the timing for the transmission of the sensor values to the host pc can be adjusted. Three different modes are available for the latter adjustment, but only one will be implemented. The first mode is used to request a sensor value over the serial interface each time an electrical impulse is sent to two pins of the device. In the second mode, a request packet is sent to the device, which then sends a new sensor value back. In the third mode, called *free running*, the device automatically sends a sensor value with a programmed frequency of up to $1\,\mathrm{kHz}$. This is the operation mode that will be implemented for the use on the experimental satellite. The software and electrical interface is built upon the design specification of the µFORS manual [9].

In every operation mode, one of two packet sizes can be chosen, as well as the range for the measured values. Both parameters have an influence on the accuracy of the sensor data. Depending on the configuration of the device a signed integer of length $16\,\mathrm{bit}$ or $24\,\mathrm{bit}$ is sent over the serial interface. Its value must be multiplied by a factor that is given in the user manual, yielding an angular rate in $^\circ/\mathrm{s}$. The accuracy of the sensor value depends on the length of the integer and the range of values it represents. For example, the $24\,\mathrm{bit}$ signed integer can adopt 2^{24} different values from -2^{23} to $2^{23}-1$. The smallest selectable range of the µFORS is $\pm 19.991^\circ/\mathrm{s}$, which result in the smallest possible representable value of

$$\Delta = \frac{2 \cdot 19.991^\circ/\mathrm{s}}{2^{24}} \approx 2.34 \cdot 10^{-6^\circ}/\mathrm{s}\,. \tag{4.1}$$

This range suffices for almost all maneuvers a typical satellite has to endure, and the accuracy of the sensor itself is not as good as the theoretical accuracy of the transmitted value.

The sensor could also output an angle increment from the last time the sensor value was requested, or the total angular rotation since the sensor was powered on, but for the experimental satellite the angular rate is the most useful type.

4.3.1 Protocol

The protocol for the data exchange between OBC and the μFORS device is very simple and is shown in Tab. 4.2. The total length of a packet depends on the chosen accuracy for the sensor value. Some housekeeping data is sent back via a *status* byte, used for notifications

Table 4.2: μFORS packet layout and corresponding field sizes in byte

sensor value	status bits	checksum
2 B or 3 B	1 B	1 B

of critical errors like temperature warnings or hardware defects. Only one packet type is needed for the communication with the device and no request or acknowledgment packet is necessary in this autonomous operation mode.

4.3.2 Special functions

Checksum

To ensure trustworthy sensor data, every packet includes a checksum byte to check for transmission errors. The algorithm for this check, like the communication protocol, is fairly simple. All four to five bytes of a packet are added together, including the checksum, and the result is limited to a byte, by omitting all digits above the eighth bit. If the resulting value does not equal 255 a checksum error occurred.

In the rare situation that such an error happens, the data packet will be discarded. This results in an incorrect attitude if the gyros are used for inertial attitude determination by integrating the angular rate over time, but the effect is very limited if it only happens rarely.

Resyncing serial communication

A problem with the free running mode is the synchronization between both serial partners. It is possible that more than or less than a whole packet is received in one instance. When writing all received data into a buffer, it is usually possible to reconstruct the start of a packet. However, if a transmission error causes the checksum to be incorrect it may be impossible to find the bounderies of the packet. For this situation a global variable is used to count checksum errors that occur consecutively. If this counter exceeds a defined threshold, chances are higher that the communication is desynchronized, than that multiple checksum errors actually occurred in a row. In that case a resyncing algorithm tries to find a complete packet in the data stream. In this situations multiple sensor values are lost, which can lead to problems with the inertial attitude determination that is solely based on the rate gyros, and an immediate shutdown of the experimental satellite and the overall simulation must be considered.

4.3.3 µFORS-6U software interface

Fig. 4.3 shows the outline for the software interface for a µFORS rate gyro. The highlighted path shows the normal process if everything works as planned. If a non-standard situation occurs, this path is left and the routine is ended or a special branch of the flow chart is executed. The timer is called by Simulink™ at a frequency chosen equal to or higher than

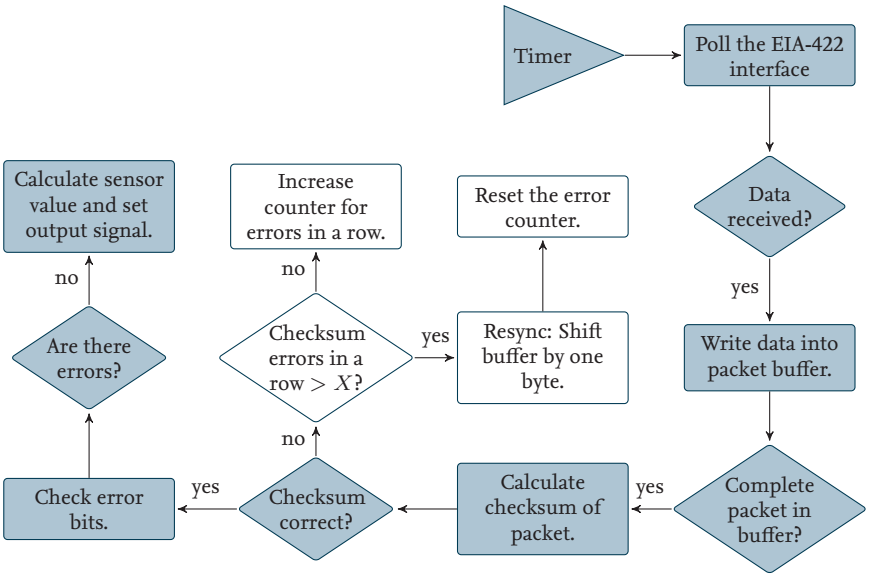


Figure 4.3: Flow chart of the µFORS software interface

the programmed frequency for the free running mode on the device. If the frequency is too small, sensor values will become outdated and the OS has to reserve more and more memory for the serial data queue.

The software routine starts with polling the serial interface. If data was received, it is written into a small packet buffer. The buffer is needed for the case that either not a whole packet is received in one run, or the synchronization is lost. After all received data is written into the buffer, it is searched for a complete packet, starting with the newest possible packet in the buffer Most of the time, the amount of data in the buffer is exactly the number of bytes in a packet, and the searching process is immediately successful. To check whether or not the packet is really a regular packet, the checksum is tested. If the checksum is valid, it is very likely that the correct bytes have been taken from the byte stream. However, the probability of the checksum being correct due to mere coincidence is not negligible. In the checksum is not valid this can have two different reasons: A transmission error has occured, or the process is desynced.

The algorithm for determining the cause of the error is the lighter branch in the middle of Fig. 4.3. If a checksum error occurs more than three times in a row, it is assumed that there is no complete packet in the buffer, and a resynchronization process is started. Otherwise, the corrupt data is deleted from the buffer and thus one sensor value is discarded.

In the resynchronizing process, only the first byte of the buffer is deleted, and the rest is

shifted by one byte every time the routine is called, until a correct packet is found in the buffer.

When a correct packet is found, the status bits are checked for any detected hardware defects; afterwards, the integer value from the packet is decoded, multiplied by the correct factor (see equation 4.1 for an example), and set as the output signal.

The source code for the software interface to the μFORS rate gyro can be found in appendix C.2 and the Simulink™ interface block is shown in Fig. C.2, in the same section of the appendix.

4.4 AMR magnetometer

The magnetometer uses an EIA-422 interface for sending the magnetic field sensor values to the OBC. Additionally, the temperature inside the device is measured and is always sent alongside with the magnetic field information.

Every sensor value must be requested separately, and the measurement does not start until the device receives the command. The duration for the measurement can vary, depending on the used request command. Different commands are used to set the frequency of the analogue-digital-converter for the magnetic field measurement, resulting in 1, 2, 4, or 6 sensor values per second [10].

Except for the temperature, no further housekeeping data is available. One specialty of the AMR magnetometer is the ability to shut down parts of the device to lower its power consumption. Only the so-called analogue part for the measurement of the magnetic field can be switched off, but the digital part for the serial interface stays powered on to restart the measurements when needed.

4.4.1 Protocol

The packet structure of the AMR device is straightforward and does not contain any method to verify the integrity of the data. Every measurement value must be requested separately by a one byte command. The bytes 0x50. . . 0x53 request the sensor values; they are sent after the above mentioned delay required for measuring the magnetic field vector and converting it to digital information. The sensor packet structure is shown in Tab. 4.3. All four sensor values are each sent as a $16\,\mathrm{bit}$ integer. The magnetic field components

Table 4.3: AMR magnetometer sensor packet layout

channel x	channel y	channel z	temperature t	command byte
$2\,\mathrm{B}$	$2\,\mathrm{B}$	$2\,\mathrm{B}$	$2\,\mathrm{B}$	$1\,\mathrm{B}$

have to be multiplied by ten to gain the field strength in nT and the temperature integer must be multiplied by the factor of 0.005519 for a value in $^\circ\mathrm{C}$.

The two commands for switching the analogue part on and off are 0x99 and 0x98. The

identical byte is sent back as an acknowledgment that the command was successfully executed by the AMR magnetometer.

4.4.2 AMR magnetometer software interface

The software interface to the AMR magnetometer has two main task to fulfill: polling the device for sensor values at different frequencies, and changing the power state according to the user requirements. Fig. 4.4 outlines the routine to control the device. As for all the

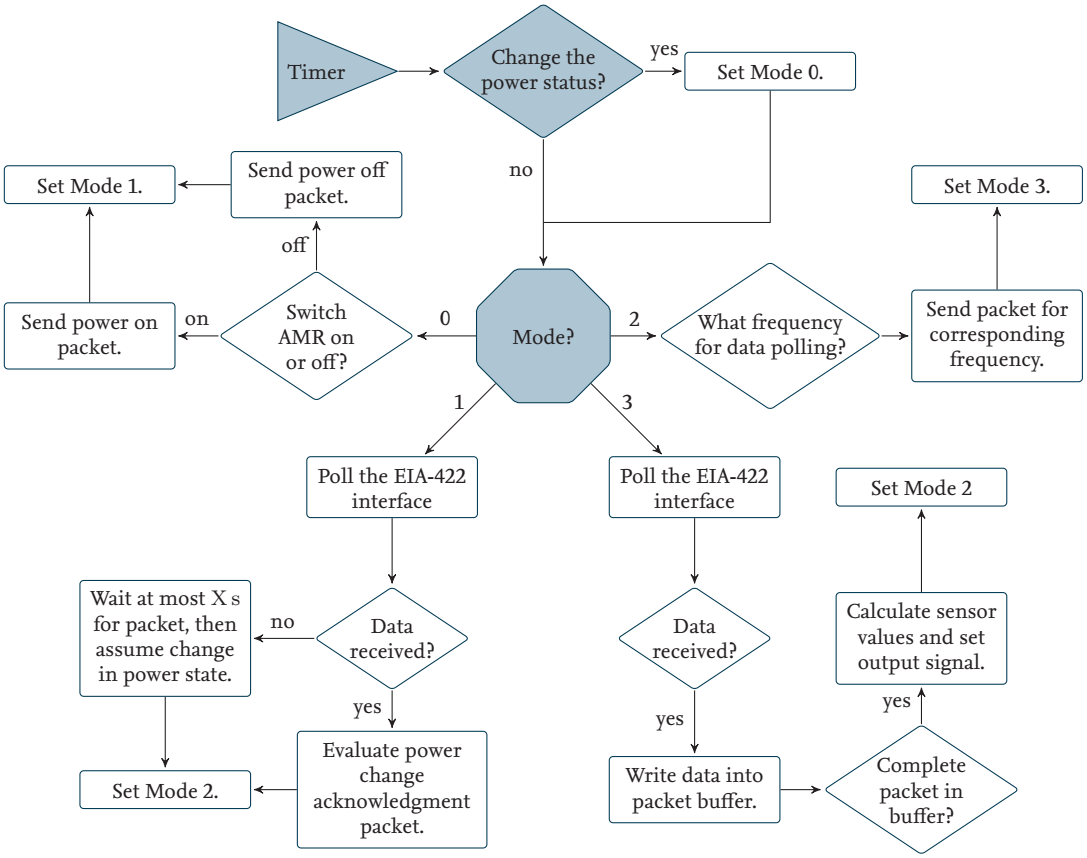


Figure 4.4: Flow chart of the AMR magnetometer software interface

devices, the routine starts when it is called by Simulink™. The time between calls can be chosen much higher compared to the RW 250 or µFORS routine, because the maximal frequency for polling sensor values is 6 Hz. It is possible to use higher frequencies than 10 Hz to 20 Hz to run this function, but it would be a waste of computation time. Simulink™ can handle different execution frequencies for different devices, meaning that a slower frequency comes with no disadvantages, except that sensor values are not available as quickly after the serial interface receives a packet. On the other hand, the magnetic field will usually change only marginally between two measurements, so a slightly late sensor value has hardly no impact.

When the function is called, the first step is to check whether the power state should be changed or not. After this, a condition splits the routine into four branches, depending on the value of a global Mode variable. The Modes 0 and 1 are for changing the power state of the analogue part of the device, and 2 and 3 for requesting and receiving sensor values.

If the user requests a change in the power state then the routine sends the corresponding command through the serial interface and exits the routine after setting the global Mode to 1. In this mode the function is waiting for an acknowledgment response sent by the magnetometer. In some rare cases it can happen that no response is sent, in particular, during the first run of the routine it is not necessarily obvious whether the analogue part is on or off. If it is off, and the command to switch it off is sent, the device does not react at all, an thus the routine will never change the Mode. To resolve this situation, a watchdog timer is started after the serial interface has affirmed sending the command. If the timer exceeds a predefined number of seconds, the routine assumes a successful power change and switches to Mode 2.

Most of the time, the Mode will be 2 or 3. If it is 2, the request packet for the desired measuring frequency is sent. After changing to Mode 3, the routine waits until the serial interface receives data. If something was received, it is written to a buffer for the unlikely event that just a part of the sensor packet was transmitted by the magnetometer at the time the serial interface was polled. Each time a byte is written to it, the buffer is checked for a complete packet of 9 B length. If enough bytes have been received, there is no choice but to trust that this is an error-free packet, because without a checksum no test can be performed. The components of the magnetic field vector and the temperature are calculated and sent to the output of the routine. Finally, the Mode is set back to 2 to request new sensor values.

The Simulink™ source code for the software interface to the AMR magnetometer is attached in appendix C.3. Fig. C.3 shows the graphical representation of this interface that can be used easily by someone who has no deeper knowledge of the protocol or the special abilities of the device.

4.5 Stepper motor controller TMCM-310

The TMCM-310 is a commercial three-axes stepper motor control board with a EIA-232 serial interface. It has implemented an extensive protocol to control stepper motors through all kinds of commands. Additionally, the commanding can happen through different protocols: a binary protocol for efficient communication, and several types of ASCII protocols that could be used for direct human interactions via a terminal client [12]. The type of protocol to be used is set in the memory of the control board. The manufacturer of the test facility has provided a program to control the stepper motors for basic interactions. However, this software does not suffice for the intended purpose of using the steppers in a control loop. It was decided to utilize the same protocol in the new implementation as is used in the provided implementation, so that both programs can be used without reprogramming the controller board.

The efficiency of the ASCII protocol, in which every character is echoed back to the software interface, is reduced even more by the tunneling device, which is used for the wireless transmission. This software implementation does not rely on a special communication link; rather, it passes the data to another subroutine that forwards it to a standard serial device or, like in this case, through an ethernet device. The ethernet interface is presented in section 4.5.4.

In the end, the software interface described in this section is only capable of commanding each axis with a frequency of about $2\,\mathrm{Hz}$, which suffices for the intended purpose. By using a different protocol, a much higher commanding rate could be achieved. Only a small portion of the complex protocol is implemented in this software interface. Each stepper axis can be commanded to a relative velocity between -2047 and $+2047$. Also, the interface can query the end stops (also called reference switches) of the fine adjustment mechanisms as to whether the maximal positions of the weights have been reached. In this situation the motors are automatically stopped by the board, and no further displacement is possible.

4.5.1 Protocol

The packet structure for the ASCII protocol is shown in Tab. 4.4. It includes all commands that have been implemented in the software interface. The three steppers are aligned with the major axes of the experimental satellite. The names of the three steppers are 0, 1, and 2 for the x-, y-, and z-axis, respectively. The variable a must be substituted with the name of the stepper, and the four digits of the velocity v must be computed and converted into ASCII code; see section 4.5.2. The answer packet for the reference switches includes the logical state s of the switch that is true if the weight of the fine adjustment mechanism is in contact with the switch, and false otherwise.

Table 4.4: TMCM-310 ASCII protocol command and answer packets

description	command packet	answer packet
Set velocity $v \geq 0$ for axis a	AROR a,$v_0v_1v_2v_3$\n	BA 100
Set velocity $v < 0$ for axis a	AROL a,$v_0v_1v_2v_3$\n	BA 100
Get left reference switch status s for axis a	AGAP 9,a\n	BA 100,s
Get right reference switch status s for axis a	AGAP 10,a\n	BA 100,s

In addition to sending the answer packet, the TMCM also echos back every character of the command packet, and it is not possible to send multiple characters at once. After sending the newline character "\n", i. e., the enter key, that completes the command, the command is evaluated by the TMCM board. The answer packet is sent as a whole and starts with a series of characters (BA 100), if the command is accepted and correct. For commands that request data back, e. g., requesting the status of a reference switch, a comma is added to the series, followed by the truth value (0 or 1).

4.5.2 Special functions

Converting integers to a series of ASCII characters

The calculation of the velocity value from an integer to four ordered ASCII characters is done with the modulus function and integer division plus a bias to use the correct ASCII code for digits (cf. appendix D). The following extract from the source code shows the implementation of this function for the integer valued velocity.

Listing 4.1: Converting an integer to a series of chars.

```
int velocity;
char v_0, v_1, v_2, v_3;
v_0 =     velocity/1000+48;
v_1 =    (velocity%1000)/100+48;
v_2 =   ((velocity%1000)%100)/10+48;
v_3 =  (((velocity%1000)%100)%10)+48;
```

The four characters, v_0, v_1, v_2 and v_3, can now be used in a packet to set the velocity of a stepper motor.

4.5.3 TMCM-310 software interface

The efficiency of the protocol is so low because a lot of time is wasted by waiting for the echoed characters. The computation time is not the problem, because when it is waiting for the response and nothing is received, the routine just exits. To some extent it is possible to increase the frequency for the execution of the routine, but at some point most of the routine calls are aborted due to the fact that no data was received. Calling the function that is outlined in the flow chart in Fig. 4.5 with about $100\,\mathrm{Hz}$ results in a commanding frequency of $2\,\mathrm{Hz}$ (per axis), including querying all six reference switches (one per end of each fine adjustment mechanism). This means that approximately 18 commands per second are possible with this implementation of the ASCII protocol.

When the timer is called by Simulink™ the routine works either as a sender or as a receiver, depending on a global variable that is switched back and forth between both states. At the beginning, a new packet has to be built, where the first and second packet in each cycle of the three axes are for querying both reference switches. The third packet to be built is to set a new velocity, based on the value of the input of the Simulink™ block. After these three packets are completely sent, the process is cycled to the next axis. Each time the function is called the routine sends exactly only one character of the packet over the serial line and exits after switching to Mode 1.

The receiving mode starts by polling the serial interface, and if data was transmitted, it has to decide if this is just an echoed character or the answer packet. An echoed character simply leads to switching back to Mode 0, in which the next datum will be sent. An answer packet, on the other hand, must be buffered for the case that just parts of the packet have been received. A completely buffered packet is processed depending on its content. The confirmation (BA 100) of an accepted command for a new stepper velocity only leads to switching to transmission mode. If a reference switch status report was received, the truth

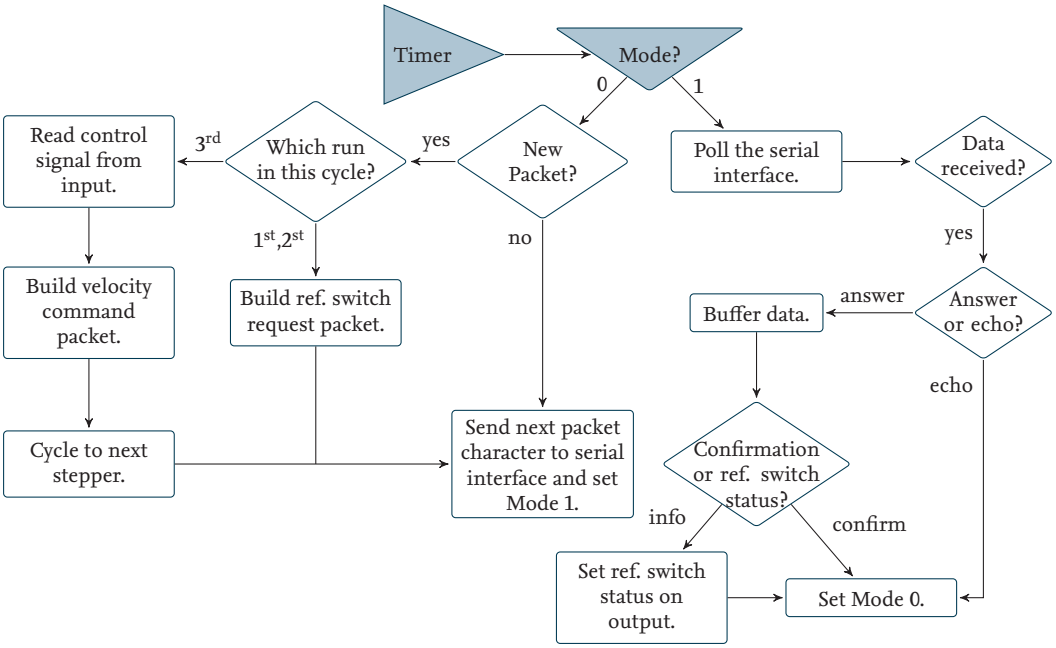


Figure 4.5: Flow chart of the TMCM-310 stepper control board software interface

value must be extracted from the ASCII series and the output of the routine must be set accordingly. Depending on the side of the reference switch and the current value of the output, it has to be decided if the status has to change. If the left switch is activated a -1 should be set on the output, a $+1$ for the right switch. In any other case the output value is zero. It should be noted that a negative response, i. e., an inactive switch, does not necessarily result in a zero on the output, because the other switch might be active.

The complete source code of the software interface for the TMCM-310 stepper motor control board can be found in appendix C.4. The graphical interface for the Simulink™ environment is shown in Fig.C.4.

4.5.4 Basic TCP/IP interface

Communication between all kinds of computers nowadays mostly happens over ethernet, also called local area network (LAN)/WLAN or just network. Ethernet is a serial communication with several protocol layers. The Transmission Control Protocol / Internet Protocol (TCP/IP) is one of the highest layers, used to transmit data between network interfaces, different computers, or other digital devices. The implementation of an interface to use this protocol in the Simulink™ environment is the subject of this section.

Every network device has its own unique identification, the IP address. The TCP/IP protocol can be used in huge networks where every device can communicate with all other. It is also possible for one computer in a network to run multiple programs, all of which can communicate with other devices in the network simultaneously. For this purpose a computer that is capable of TCP/IP connections has multiple inputs and outputs that are

called *ports*. A port can only be used by one single program on a computer. If two programs on different computers want to exchange information, one program must open a port on its host computer, to which the other computer connects using the known IP address and port number.

At the moment, this interface is needed for the fine adjustment mechanisms from section 4.5.3, which are connected to a WLAN device. Possible future applications include connections to the FACE control computer or other devices not mounted on the experimental satellite. While the FACE can only simulate the rotational freedoms using the air bearing table, the translational freedoms could be simulated with an orbit propagation software on an external computer. This simulation data could be provided to the OBC in real-time as a pseudo Global Positioning System (GPS) signal, containing location and velocity information. The TCP/IP interface could be used to handle underlying communication.

4.5.5 TCP/IP software interface

The source code for the interface is adapted form several tutorials and examples for POSIX sockets programming. Sockets are ports from the point of view of the programming language. They have to be created by a server application to which a client can connect. The server then acknowledges or declines the connection. The software interface for the Simulink™ environment can either represent a server or a client. If a server should wait for another application to connect, the desired port on which the server is to listen should be given as a parameter, whereas a client needs to know the IP address and the port number of the server to which to connect.

Exchanging data in Simulink™ is very easy with this interface. After the connection is established, the interface only needs information about the amount of data to receive or to send. The data that should be sent to the other communication partner must be passed to the TCP/IP interface, and the received data is available via the output of the routine. The graphical interface is shown in Fig. C.5 and the adapted source code for the Simulink™ environment is given in appendix C.5.

5 Initial operations and tests

5.1 Adjusting the experimental satellite's center of mass

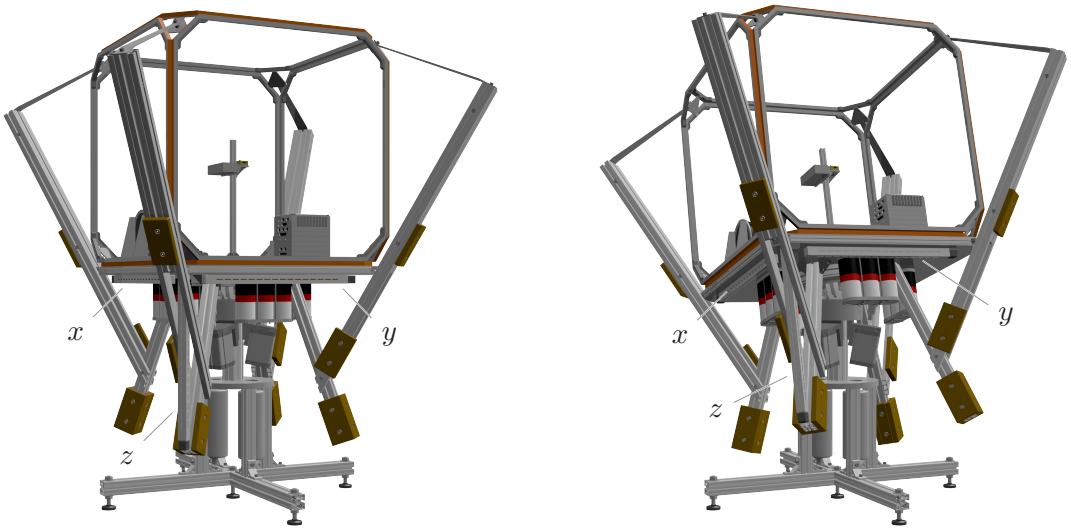
The c. m. has to be adjusted very well for the experimental simulation of the attitude dynamics of a free floating satellite in space. In absence of gravity or its cancellation through centripetal forces, a satellite would always rotate around its c. m. no matter where it is located. The experimental satellite rotates around the pivot point of the air bearing. If the c. m. and the pivot point of the bearing differed, a torque would pull the assembly station to a zero position and induce an oscillation around it or the assembly station would tip over.

The consequence of this behavior is that the c. m. has to be moved towards the pivot point, and, if possible, this should be done automatically every time there is a change in the setup of the experimental satellite. To automate this procedure, the fine adjustment mechanisms are used. Before these small displaceable weights of $100\,\mathrm{g}$ can be used for the adjustment, the c. m. has to be tuned coarsely by moving the large weight masses on the assembly's beams manually. In the beginning this is a trial-and-error based task to find a position where the table is stable and does not tip over or hit the stops at the maximal deflection angles. After such a rough position is found, very small change in the weights' locations are necessary to locate the c. m. somewhere below the pivot point. At this point the system should be very stable, and the weights can now be symmetrically moved upwards until they reach the point where the table is still stable, but a fine adjustment in the z or gravitational direction suffices to render the table unstable.

After this coarse adjustment the pneumatic actuators from Fig. 3.3 are released and the table is held in a well defined attitude. In this resting position the assembly station is aligned with a precision water level by adjusting the feet of the support stator. This guarantees that the fine adjustment mechanisms in Fig. 5.1(a) for the x and y direction have no influence on the gravitational direction for which only the z mechanism shall be responsible.

The automatic balancing of the assembly station with the experimental satellite has some requirements that will be discussed in detail later. First, an attitude controller is needed that uses the reaction wheels and the fiber optic gyros. The control algorithm is a very basic proportional-derivative (PD) controller that is specially designed for the platform's estimated moment of inertia. Also, an attitude determination is required that eliminates the angular rate of the Earth's rotation and integrates the rates' components to obtain a complete attitude. The necessary reference frame descriptions can be found in appendix A. Fig. 5.2 shows the control diagram for this problem.

The neutral position of the platform (assembly station and experimental satellite) is defined by the released (extended) pneumatic actuators. After the attitude controller is en-



(a) Adjusting the c. m. in the x-y-plane

(b) Adjusting the c. m. in the z-direction

Figure 5.1: Fine adjustment mechanisms to influence the c. m. in all three major axis

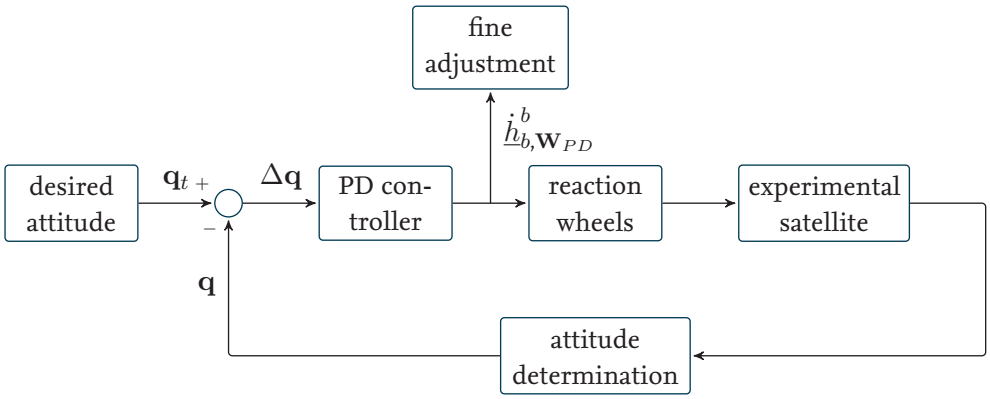


Figure 5.2: Control diagram for the automated fine adjustment process

abled, the pneumatic actuators are retracted, and from this point on the platform is only supported by the air bearing. The off-centered c. m. induces a torque which the reaction wheels have to balance in order to hold the attitude. Depending on the magnitude of the torque, the wheels could start saturating rapidly. The torque $\dot{\underline{h}}^{b}_{b,\mathbf{w}_{PD}}$ calculated by the controller accelerates the flywheels and compensates for the misaligned c. m. (cf. Fig. 5.2). This torque signal is simply used to control the fine adjustment mechanism in the way that a torque around the x axis is balanced by displacing the y adjustment mass and vice versa.

The torque caused by the off-centered c. m. is small and the adjustment weights can be moved quickly enough, then the control torque of the reaction wheels will decrease and at

some point no changes in the angular velocity of the wheels are visible. When the accuracy of the adjusted c. m. meets a predefined condition, the stepper motors are shut off and the basic ACS will target the next position where only the z adjustment will be used.

The different positions for the automated process can be seen in Fig. 5.3. The three

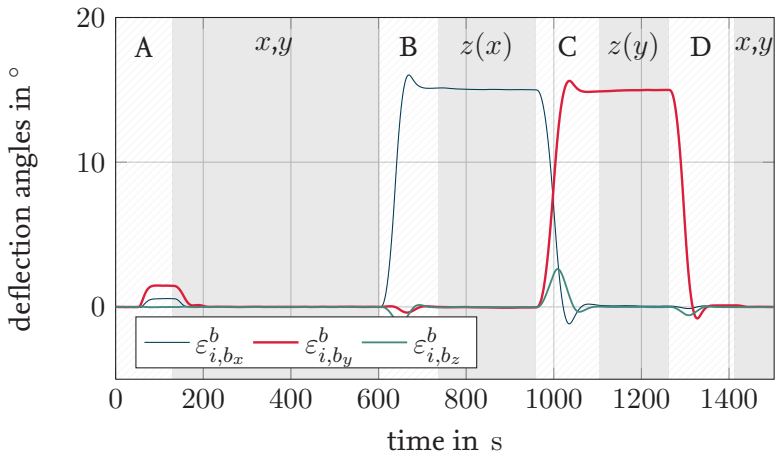


Figure 5.3: Deflection angles of the experimental satellite during the automatic adjustment of the c. m.

curves show the course of the deflection angles around the body fixed axes. In the hatched areas of the diagram, A, B, C, and D, the ACS approaches a new attitude. The first attitude is defined by the body fixed deflection angles of zero for the x and y axis. Attitude B is deflected around the x axis by $15°$ and C is a rotation around the y axis, also by the amount of $15°$. The last attitude D is identical to A. At the end of each maneuvering phase the ACS has almost reached a steady state for the attitude.

During the first $50\,\mathrm{s}$ the pneumatic actuators are released and hold the platform in its reference position. By retracting the actuators, the platform starts moving due to the off-centered c. m., which the ACS tries to compensate through the reaction wheels.

After a maneuver ends and the angular rates are almost zero, the fine adjustment mechanisms are used to reduce the control torque commanded to the wheels. The shaded areas of the diagram show the periods of time in which stepper motors are active; the controlled axes in each particular period are indicated at the top. In the first and fourth interval, the x and y weights are moved based on the control torque for the y and x axes. A simple P controller calculates the stepper velocity $\underline{v}^a_{a,s}$ (with respect to the assembly frame a, cf. appendix A.3) with empirically assigned gains in the following control laws:

$$A \text{ and } D: \quad v^a_{a,s_x} = 6000 \cdot \underline{\dot{h}}^b_{b,\mathbf{W}_y} \qquad v^a_{a,s_y} = 6000 \cdot \underline{\dot{h}}^b_{b,\mathbf{W}_x} \tag{5.1}$$

$$B: \quad v^a_{a,s_z} = 6000 \cdot \underline{\dot{h}}^b_{b,\mathbf{W}_x} \qquad C: \quad v^a_{a,s_z} = 6000 \cdot \underline{\dot{h}}^b_{b,\mathbf{W}_y} \tag{5.2}$$

The z adjustment mass is displaced in the second and third interval, depending on the control torque of the deflected axis in these maneuvers, whereas the torque of the reaction wheel for the z axis is never used to control the weights.

After the first maneuver is finished it is known that the c. m. must be located above or below the pivot point, because (almost) no control torque is necessary to hold the steady state of the experimental satellite. To adjust the c. m. to coincide with the pivot point, the experimental satellite is deflected by $15°$ as shown in Fig. 5.1(b). The effect of rotating the satellite is that the c. m. is also rotated alongside, and once again a disturbance torque caused by the gravitational force acts on the system. The fact that the c. m. is correctly located in the body fixed x-y plane is exploited in this maneuver in the way that only the fine adjustment mechanism for the z direction has to be used to move the c. m. towards the pivot point.

These assumptions are of course a simplification of the real process, because the mechanical tolerances of the assembly station allow small errors in the alignment of the three mechanisms towards each other and therefore the displacement of the z weight has a small influence on the c. m. in the body fixed x-y plane.

The first three intervals of fine adjustment maneuvers from Fig. 5.3 are shown in more detail in Fig. 5.4 with respect to their body fixed control torque $\underline{\dot{h}}^{b}_{b,\mathbf{W}}$, stepper velocity $\underline{v}^{a}_{a,s}$ and error angles $\underline{\varepsilon}^{b}_{g,b}$ towards their desired guidance attitude g of the four phases. Because the velocities of the steppers are based directly on the control torque of the ACS multiplied by a proportional gain, they have a similar shape, unless the steppers are shut off. Especially at the beginning of the first fine adjustment phase ($130\,\mathrm{s}$) between maneuver A and B, a constant offset towards the desired attitude is visible, $-0.5°$ in x and $-1.5°$ in y direction. The constant offset of the steady state (with respect to the desired attitude) before the steppers are turned on, is the result of using a PD controller that is not able to compensate for a directed disturbance torque [8].

After the fine adjustment mechanisms start moving the weights, the offset decreases very rapidly. To avoid the risk of an unstable reaction from the ACS the velocity is limited to ± 30, which is very slow. After a few oscillations in all three diagrams for the x,y adjustment, a new steady state is reached that has no offset towards the desired attitude and almost no control torque is commanded by the ACS to hold the attitude, besides some noise on all signals.

In the following two adjustment phases a similar reaction can be seen, except that the error angles before starting the steppers are smaller by an order of magnitude.In these cases, only the z adjustment is steered by the control torque of the x or y reaction wheel, depending on the deflected axis. At the end of both phases, all curves are close to zero, which indicates a well adjusted c. m. The last maneuver D rotates the experimental satellite back to the initial position where the adjustment for the x,y direction is repeated. This is done because a misaligned z adjustment axis may have moved the c. m. in the x-y plane. At the end of the whole adjustment process, the experimental satellite has a well tuned c. m. very close to the pivot point of the air bearing, especially at the home position, i. e., the initial attitude, where all experiments start after the pneumatic actuators are retracted.

5.1.1 Process review and future improvements

To put the accuracy of the automated adjusting of the c. m. into perspective, the attitude controller was shut off after the process finished. Between $1500\,\mathrm{s}$ and $1800\,\mathrm{s}$ in Fig. 5.5,

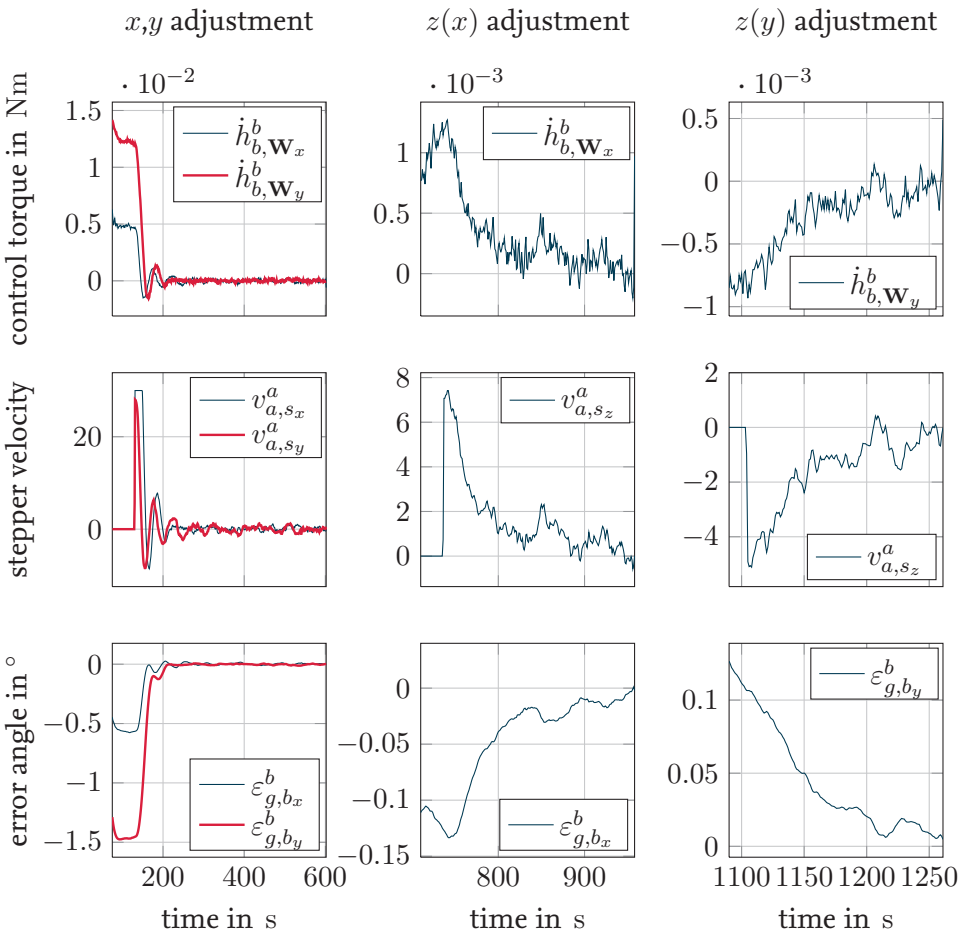


Figure 5.4: Attitude and control information during the adjustment phases

the experimental satellite floated without any interactions from the ACS. The deflection angles ε slowly grew to larger angles, whereas the angular rates ω oscillated with a small tendency towards higher rates. This shows that the c. m. was not precisely adjusted and the air bearing table seemed to be slightly unstable, thus the satellite tipped over slowly due to the off-centered c. m.

At $1800\,\mathrm{s}$ a small piece of paper, about $4\,\mathrm{cm}^2$ ($80\,\mathrm{g/m}^2$), was dropped on top of the assembly station, $30\,\mathrm{cm}$ away from the center. Over the course of the next $100\,\mathrm{s}$ (the shaded area in the diagram), the absolute value of the angular rates $\omega^b_{i,b}$ more than doubled its value, and the total deflection angle $\varepsilon^b_{i,b}$ of the experimental satellite increased considerably.

This small test shows that the adjustment process does not result in a perfectly aligned c. m.; nonetheless, a small piece of paper had a definitively measurable influence on the experimental satellite with moments of inertia of approximately $34\,\mathrm{kgm}^2$ around all its major axes. This shows how well the assembly station was tuned.

To improve the results of the process, the two separate phases for the displacement of

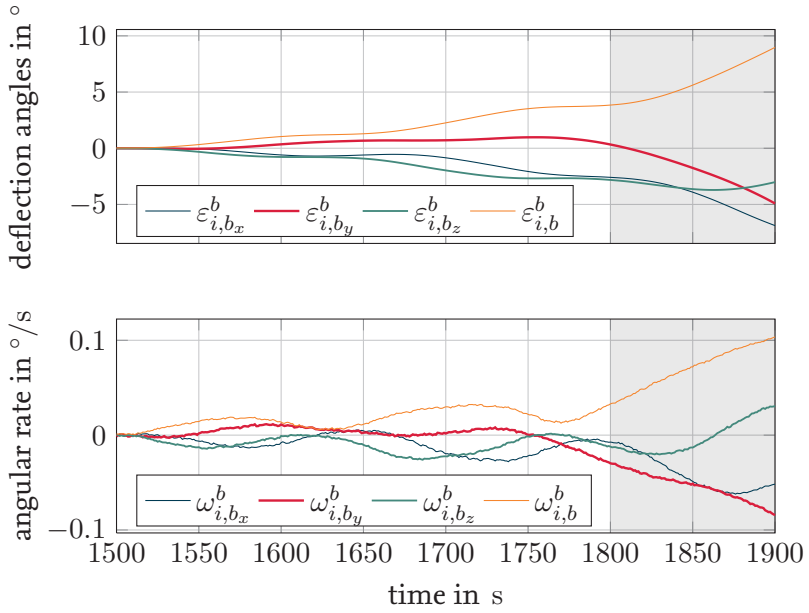


Figure 5.5: Free floating experimental satellite

the weight in z direction should be merged into one phase, or an average position based on both phases must be found. Currently the stop criterion for an adjustment phase is that the stepper velocity does not exceed a certain threshold over a given period of time. For example, the threshold for the x,y adjustment is the steppers smallest possible velocity of 1, which must not be exceeded for a period of 60 s. Possibly a better, or at least a faster, criterion can be found to determine a well adjusted phase. Another improvement might be to calculate the optimal gains for the control torque instead of using the empirically chosen gains used for the velocities of the steppers (cf. 5.2). For safety reasons, the gains were chosen very small to avoid problems with the stability of the control loop. In the case of a highly misaligned c. m., this algorithm is too slow to correct the alignment before the reaction wheels saturate.

Almost every day the test facility is used, it is necessary to repeat the auto adjustment, even if no change in the setup of the experimental satellite was made; sometimes even twice a day, if the temperature differences are very high. This shows how important it is to reduce the duration of this process. Finally, there should be a way to characterize how accurate the fine adjustment process was, in order to give an estimate for a lower bound of the possible duration of an experiment, before the slowly accelerating reaction wheels saturate through the misaligned c. m. and the experiment has to be aborted.

5.2 Measuring and adjusting the experimental satellite's moment of inertia

An attitude control system for maneuvers with high precision demands a well known satellite's moment of inertia. This is important in the controller design, the feed forward control, and the attitude determination, if, for example, a state estimator with a prediction model is used. The higher the requirements for the ACS are, the more precisely the moment of inertial has to be known. Small and slow maneuvers do not necessarily require this if a robust approach for the controller design is chosen. The attitude controller for the fine adjustment process from sec. 5.1 is an example for a robust design, where an estimated moment of inertia tensor is used to control the experimental satellite. On the other hand, if having a high guiding accuracy or reaching a fast steady state is required without overshooting the target several times, a good calculation or measurement of the moment of inertia tensor is essential.

For the experimental simulation of different satellites it is also important to be able to adjust the inertia tensor, to verify an ACS with the expected mass distribution of the designed satellite.

5.2.1 One dimensional moment of inertia

Before the inertia tensor was measured for the whole assembly station and the attitude control hardware, some tests were done for the measurement of the one dimensional moment of inertia around the z axis of the experimental satellite. The configuration of the air bearing table was stable with a c. m. below the pivot point and only one reaction wheel and one gyro were operational for the vertical axis, which, contrary to the other axes, does not possess any limitation in its degree of freedom. This way the test can be performed in a safe environment without the risk of tipping the table over or getting near to the mechanical limits at the maximal deflection angles of the x and y axis that prevent damage to the bearing.

Change in angular rate

The first attempt at measuring the moment of inertia around the vertical axis of the air bearing table was based, similar to later attempts, on the concept of conservation of angular momentum. The measurements of the rate gyro and the reaction wheel were both filtered by a low pass that averaged the measurements over several time steps to gain clearer and less noisy signals. At the end of the test it was necessary to review the data to single out some time intervals that were too noisy or clearly wrong.

The computation was based on equation 2.37; as a simplification, the angular rate was assumed to be parallel to the angular momentum of the experimental satellite, causing the second term of the equation to vanish. The change in angular momentum is written in terms of the angular acceleration, and the (yet unknown) body fixed moment of inertia

tensor (cf. equation 2.29). The one-dimensional equation for this problem is

$$I^b_{\mathbf{B}} \frac{\mathrm{d}_b \omega^b_{i,b}}{\mathrm{d}t} = T^b_{i,b} = -\dot{h}^b_{b,\mathbf{W}} \Rightarrow I^b_{\mathbf{B}} = \frac{-\dot{h}^b_{b,\mathbf{W}}}{\frac{\mathrm{d}_b \omega^b_{i,b}}{\mathrm{d}t}} \,. \tag{5.3}$$

The angular acceleration was calculated using the numerical derivative of the rate gyro's measurement and the reaction wheel was used to apply a chosen reaction torque to the system. The moment of inertia of the flywheel is known from the manufacturer, and the angular velocity is accessible via the requested housekeeping data of the device. The angular velocity was numerically derived as well, yielding the actually applied torque $\dot{h}^b_{b,\mathbf{W}}$ (which can differ from the commanded value).

Because the angular momentum of the whole setup is conserved, the torque generated by the flywheel equals the negative torque that accelerates the experimental satellite. The first diagram in Fig. 5.6 shows long periods of constant torque of the wheel that was manually commanded from a remote computer in a random pattern. The peak at about $330\,\mathrm{s}$ is probably related to the noisy numerical derivative after a new value was commanded to the reaction wheel, which is amplified by the low pass over several seconds. The second

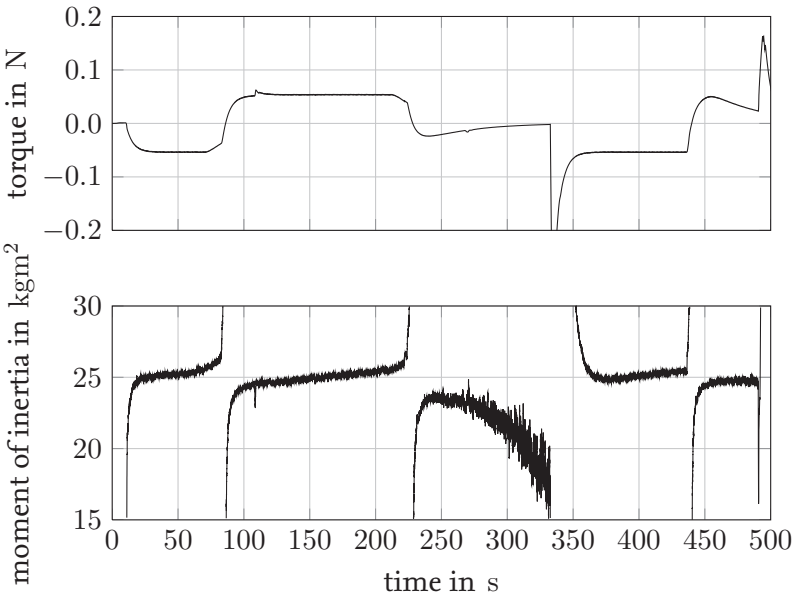


Figure 5.6: Online calculation of the moment of inertia

diagram shows the calculated moment of inertia in each time step. The peaks between two subsequent intervals are caused by small angular accelerations of the satellite that lead to divisions with very small denominators in equation 5.3. The division of the two noisy signals cannot be trusted in the surrounding of zero and should therefore be ignored. Intervals with a constant torque clearly show a more plausible value for the moment of inertia, which indicates that it is possible to estimate the inertia tensor using reaction

wheels, based on the conservation of angular momentum. The accuracy of this measurement, and all following, depends directly on the accuracy of the given moment of inertia of the flywheel inside the reaction wheel's casing.

Total angular momentum

Another idea to estimate the moment of inertia is to use the total angular momentum of the reaction wheel and exchange it with the test platform by accelerating the flywheel to a constant angular velocity. Equation 2.29 is adopted to describe the system of the experimental satellite and the flywheel.

$$h^b_{i,\mathbf{B}} + h^b_{b,\mathbf{W}} = I^b_{\mathbf{B}}\omega^b_{i,b} + I^b_{\mathbf{W}}\omega^b_{b,w} \stackrel{!}{=} \text{const} \tag{5.4}$$

This one dimensional equation states that the total angular momentum of a system is constant and with measurements from two points in time it is possible to calculate the unknown moment of inertia $I^b_{\mathbf{B}}$. For the two points in time, t_1 and t_2, and the corresponding angular momenta, the following formula applies:

$$h^b_{i,\mathbf{B}}(t_1) - h^b_{i,\mathbf{B}}(t_2) = h^b_{b,\mathbf{W}}(t_2) - h^b_{b,\mathbf{W}}(t_1) \tag{5.5}$$

This equation can be solved for the moment of inertia with the definition from equation 2.29:

$$I^b_{\mathbf{B}} = I^b_{\mathbf{W}} \frac{\omega^b_{b,w}(t_2) - \omega^b_{b,w}(t_1)}{\omega^b_{i,b}(t_1) - \omega^b_{i,b}(t_2)} \tag{5.6}$$

A series of measurements was done to test this approach, and table 5.1 shows the results. Care must be taken with the different units of measurements for the angular velocities.

Table 5.1: Measurement of the moment of inertia based on the total angular momentum

$\omega^b_{b,w}(t_1)$ in rpm	$\omega^b_{b,w}(t_2)$ in rpm	$\omega^b_{i,b}(t_1)$ in $^\circ/\text{s}$	$\omega^b_{i,b}(t_2)$ $^\circ/\text{s}$	Moment of inertia in kgm^2
0	6447	-0.01	-8.77	24.29
6448	-6446	-8.60	9.24	23.85
-6446	6448	0.08	-17.05	24.84
6448	0	-16.8	-7.50	22.88

Comparing these results with the previous approach in Fig. 5.6 shows similar but slightly smaller values for the moment of inertia. Possible explanations of this may be that the values were not transcribed correctly in the manually conducted measurement series, or that the simplification of an uniaxial description was not sufficient. Nevertheless, both tests show results in similar ranges with a margin of error of roughly $10\,\%$, which should suffice for the final moment of inertia tensor.

The two described approaches are basically identical. The difference of the satellite's angular velocities is equal to the numerical derivatives in the last section, if t_1 and t_2 are chosen very close to each other. The same applies for the torque induced by the reaction wheel, which is proportional to its change in angular velocity.

The measurable properties of the system, by the rate gyro and the reaction wheel, favor the second approach, because the sensor values can be used directly without further processing, and before and after a maneuver a steady state is available for measuring this state. But for the estimation of the three-dimensional moment of inertia, the second approach is not applicable, because two axes of the air bearing table are strictly restrained in their degree of freedom, hence it is not possible to accelerate the table to these high angular rates, let alone have intervals of a steady state for an accurate measurement process.

5.2.2 Measuring the three-dimensional moment of inertia

For determining the complete moment of inertia a lot of measurement data is necessary to reduce the influence of noise from the sensors. The data should be recorded in intervals with high angular rates as well as high angular acceleration. The control process for such requirements must be automated, because the process of determining the moment of inertia should be repeatable with the least possible effort, and it is very difficult to control the experimental satellite manually by commanding the reaction wheels. To guarantee safe usage of the platform, the motion for this process must be restrained to the technical limits of the air bearing to avoid damage to all sensitive devices.

For the automated control of the table some prerequisites are needed, just like for the fine adjustment in section 5.1. This includes a complete and robust ACS for attitude determination and control. It has to be robust because the moment of inertia itself is essential for the design of an ACS, and throughout the automated process the experiment must remain stable and controllable.

The procedure is based on approaching different attitudes successively as fast as responsibly possible, and if doable, all body fixed angular rates should change rapidly. The implementation of this procedure also requires a guidance that plans ahead and estimates the maximal possible rates in order to arrive very close to the target attitude without exceeding it and to exhaust the whole range of the air bearing without the risk of bumping into the mechanical end stops.

The evolution of the body fixed angular rates measured during the automated program are depicted in the first diagram of Fig. 5.7. Even though the reaction wheels were commanded with the maximal possible torque of $0.1\,\mathrm{Nm}$, the rates never exceeded $2^\circ/\mathrm{s}$. By choosing different target attitudes it would be possible to reach higher rates for the vertical z axis, but not for the other two. The maximal rates are not comparable to those from table 5.1, because the setup of the experimental satellite has changed in major ways. First, the attitude control hardware for all three axes were mounted onto the assembly station, and second, several weights had to be placed on the beams in Fig. 3.4 to move the c. m. towards the pivot point of the bearing.

The second diagram in Fig. 5.7 shows the angular momenta of the three flywheels. They were saturated prior to the experiment to amplify the gyroscopic coupling torques that

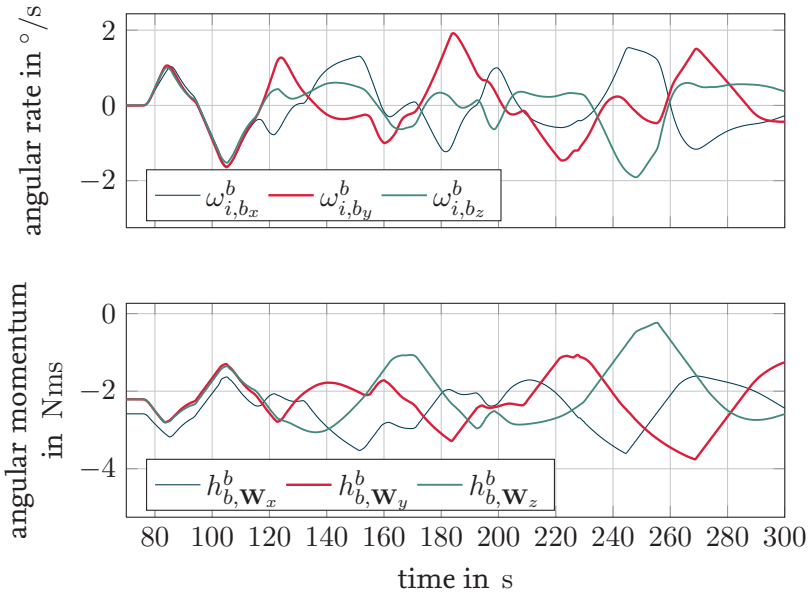


Figure 5.7: Profile of the automated program for the estimation of the satellite's moment of inertia tensor

originate from gyroscopes with externally induced torques around a non-spinning axis.

The measurement data shown in the diagrams is used to calculate the moment of inertia tensor $\underline{\underline{I}}^b_{\mathbf{B}}$ of the whole setup. This requires a mathematical model that can very accurately describe the dynamics. For this purpose, equation 2.42 is used with two modifications: No external disturbance torques $\underline{T}^b_{i,b}$ are present, and the three reaction wheels are combined into one angular momentum $h^b_{b,\mathbf{W}}$. After rearranging the formula, the model for estimating the moment of inertia tensor looks as follows:

$$\underline{\underline{I}}^b_{\mathbf{B}} \frac{\mathrm{d}_b \underline{\omega}^b_{i,b}}{\mathrm{d}t} + \underline{\omega}^b_{i,b} \times \left(\underline{\underline{I}}^b_{\mathbf{B}} \underline{\omega}^b_{i,b} \right) + \underline{\omega}^b_{i,b} \times \underline{h}^b_{b,\mathbf{W}} + \frac{\mathrm{d}_b \underline{h}^b_{b,\mathbf{W}}}{\mathrm{d}t} \stackrel{!}{=} \underline{0} \tag{5.7}$$

All quantities in this equation are known except for the satellite's inertia. For each time step, the derivatives of the sensor signals were calculated as the average of the change towards from the last and to the next value in the time series. The measurement data was smoothed using a sliding average over $0.2\,\mathrm{s}$ and reduced to one data point every $0.2\,\mathrm{s}$. This was done because the reaction wheels send new values with a frequency of $8\,\mathrm{Hz}$, unlike the gyros that use a frequency of $100\,\mathrm{Hz}$. The filtering removes any outliers from the sensor signals that are caused by the discrete sampling rate, leading to smoother numerical derivatives of the measurement series.

The moment of inertia tensor has a symmetric matrix representation of size 3×3 with six independent values that must be identified [1]. The moments of inertia are the diagonal

elements of the matrix, whereas the other elements are called products of inertia:

$$\underline{\underline{I}}^b_{\mathbf{B}} = \begin{bmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{xy} & I_{yy} & I_{yz} \\ I_{xz} & I_{yz} & I_{zz} \end{bmatrix} \tag{5.8}$$

For a satellite body with primarily symmetrical properties the diagonal elements are expected to be much grater than the products of inertia.

Equation 5.7 has three dimensions that can be used to calculate the six unknown elements of the moment of inertia. In theory, two different data points from the profile (or six, if counting the predecessor and successor needed for the derivatives) are sufficient to determine these unknowns. The problem is to find data points where all quantities influence the dynamics of the satellite. The relatively small products of inertia should not be lost in the noise of the sensors, hence the trajectories in Fig. 5.7 were chosen with very rapid changes in the angular rates, and the reaction wheels were saturated prior to the recording of the experiment in order to amplify all gyroscopic effects. By using almost all data points from the recorded series, the resulting moment of inertia is an average over the whole profile, and thus a good estimate of the mass distribution of the experimental satellite.

Solving this problem is approached as an optimization. The six elements of the inertia matrix from equation 5.8 are used as optimization parameters that are evaluated with a cost function. For this function the dynamic model from equation 5.7 is used, which is already reorganized to show the optimal result of a zero vector. The closer the optimized values come to the actual values, the smaller the error is. The problem is not unique if only one point of the measurement series is used; two would suffice, but all points of the series are taken into account. An appropriate guess for the initial value of moment of inertia matrix is used to compute the accumulated error of all points in the time series from t_{begin} to t_{end}:

$$c(\underline{\underline{I}}^b_{\mathbf{B}}) = \sum_{t_{\text{begin}}}^{t_{\text{end}}} \left| \underline{\underline{I}}^b_{\mathbf{B}} \frac{\mathrm{d}_b \underline{\omega}^b_{i,b}}{\mathrm{d}t} + \underline{\omega}^b_{i,b} \times \left(\underline{\underline{I}}^b_{\mathbf{B}} \underline{\omega}^b_{i,b} \right) + \underline{\omega}^b_{i,b} \times \underline{h}^b_{b,\mathbf{W}} + \frac{\mathrm{d}_b \underline{h}^b_{b,\mathbf{W}}}{\mathrm{d}t} \right| \tag{5.9}$$

The result of the cost function $c(\underline{\underline{I}}^b_{\mathbf{B}})$ is a scale of how well the guessed moment of inertia fits the measured data; the smaller the better.

The standard Nelder-Mead (simplex) method was used as the optimizer for this problem. It is a very robust algorithm for optimizing problems with a small number of optimization parameters, which works very well with this problem. The initially chose values for the three moments of inertia were $30\,\mathrm{kgm}^2$; a value of $1\,\mathrm{kgm}^2$ was chosen for the products of inertia. Several tests showed that the initial values have almost no influence on the result of the optimization, except if unrealistically high values are used for the products of inertia.

At the end of the optimization process, the resulting moment of inertia matrix was used to plot the costs of each data point in the optimization interval from $80\,\mathrm{s}$ to $250\,\mathrm{s}$. Even though the optimizer found a minimum, the error is significant and not constant over

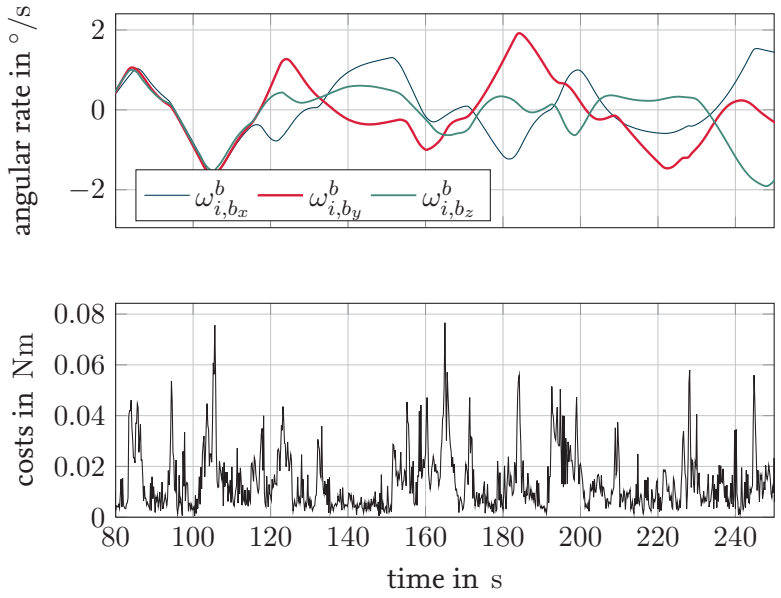


Figure 5.8: Error of the calculated moment of inertia towards the modeled satellite dynamic

time as Fig.5.8 shows. The recorded measurements are of course noisy, deteriorating the quality of the computed derivatives, but some of the peaks can also be correlated to maneuvers in the profile, where the change in the angular rate or the angular momentum of a wheel is very small. Multiplication by a small noisy sensor signal has a similar impact on the result as the previously stated problem regarding the division by such a signal.

This observation led to the idea that additionally a timing problem might be responsible for the peaks. The angular rate in the housekeeping data of the RW 250 is updated with about $8\,\mathrm{Hz}$, but the precise point in time at which a value was valid is not known. If the two signals do not correlate to each other, the mathematical model cannot accurately represent the measurement data; for example, one sensor might indicate no change in its state, whereas the other still sends varying values, contradicting the basic physical behavior of equation 5.7. Testing this idea, the measurement data from the reaction wheels and the gyros were shifted against each other by a variable time delay Δt. a positive delay projects the reaction wheel housekeeping data to a point in timer later than the moment when they were actually received by the OBC.

For each time delay, the optimization was repeated; the average costs over the recorded data is shown in the Fig. 5.9. The minimum of this curve is about $25\,\%$ lower than the mean of unshifted measurement data. The corresponding time series of the two highlighted points, the unshifted and the minimal average, are shown in the next diagram 5.10. Most of the time, the shifted measurements have a significant smaller cost than the original data. The peaks in particular show a decreased deviation, meaning that the inertia tensor in this time series more accurately satisfies the satellite dynamic as modeled in equation 5.7.

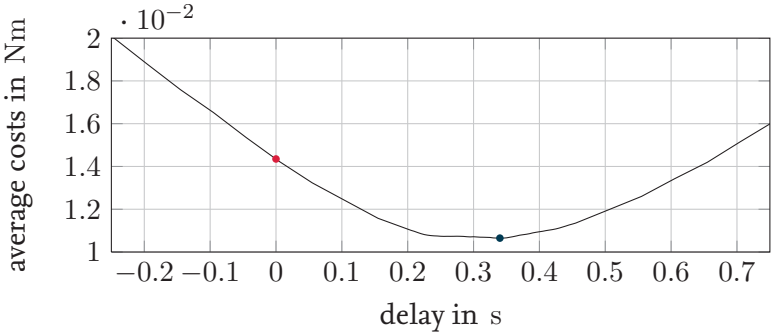


Figure 5.9: Average costs of sensor signals shifted in time

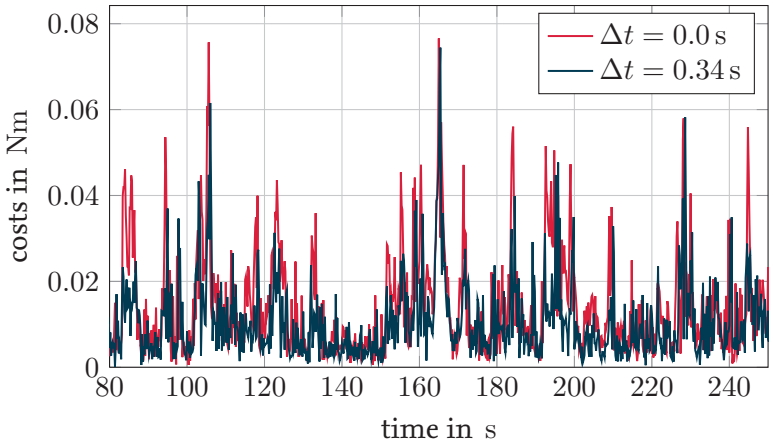


Figure 5.10: Comparison of the unshifted and the signal with minimal average costs

The implication of this finding is that the RW 250 estimates its angular velocity for a futur point in time. The user manual [8] of the device does not indicate this behavior, but because the results are more plausible after the shift, the calculated moment of inertia matrix of the experimental satellite in this configuration is

$$\underline{\underline{I}}^{b}_{\mathbf{B}} = \begin{bmatrix} 34.21 & -00.99 & -00.55 \\ -00.99 & 34.04 & 00.08 \\ -00.55 & 00.08 & 31.33 \end{bmatrix} \text{kgm}^2 \,. \tag{5.10}$$

The elements in the main diagonal should be more accurate than the products of inertia, but without any further measurement methods the absolute accuracy of these values is difficult to estimate. During all optimizations and earlier approaches the moments of inertia never deviated more than $\pm 5\,\%$ from the values given above, whereas the products of inertia already differed by $20\,\%$ in the two case addressed above.

Future modifications of this process or new algorithms for determining of the moment of inertia should focus on a reliable estimation that is based on other measurement methods with known precisions. Another possibility is the generation of a geometrical model

of the assembly station including all attitude control hardware. With such a model the moment of inertia could be approximated with high accuracy, depending on the amount of detail in the model.

If a very precise measurement is critical in the design of an ACS, it may be possible to develop a recursive algorithm that converges over several cycles. A guidance profile, like the one used in this section, can be used to evaluate the control accuracy of a controller that is optimized for a specific moment of inertia. If the controller is correctly designed, theoretically no control deviation towards the guidance profile should occur if the actual moment of inertia is known. All deviations from the guidance indicate an error in the assumed moment of inertia, which can be used to improve the estimate. The corrections could be computed directly, or an optimizer can be used to adjust the moment of inertia. Afterwards, the experiment can be repeated with updated values until the required guidance accuracy is reached.

5.3 Adjusting the moment of inertia

The weights mounted onto the beams of the assembly station in Fig. 3.4 are not only used to coarsely adjust the c. m.; they can also be used to change the moment of inertia of the test platform to realistically simulate a satellite mission with the designed mass distribution. A straightforward method would be to equally move the upper weights and the lower weights away from the c. m. to increase the moment of inertia, or to decrease it by moving them in the opposite direction. Afterwards, the c. m. must be readjusted; then the new moment of inertia could be measured. Given a required value for the mass distribution, an appropriate distance for the displacement could be calculated using geometrical information of the assembly station. Additionally, new weights with different masses could be mounted evenly on predefined locations on the beams, or the attitude control hardware could be moved along the grid on top of the assembly station. The second approach would change the moment of inertia without a change in the total mass of the experimental satellite, which is advantages because the supporting air bearing only allows a maximal load of $180\,\mathrm{kg}$. The preferred method is displacing the already present weights by a predetermined amount, reducing the effort to readjust the c. m. for a safe configuration.

6 Demonstration and verification

This chapter the basic capabilities of the Facility for Attitude Control Experiments (FACE) are demonstrated. This requires the attitude control system (ACS), parts of which were already used in the previous chapter for the fine adjustment of the c. m. (5.1) and the measurement of the moment of inertia (5.2). The first step is the development of an *attitude determination* that is based on the fiber optic gyroscopes from sections 3.2.4 and 4.3. The *attitude control* is done by the reaction wheels introduced in sections 3.2.2 and 4.2. For the actuators, multiple PID controllers have to be designed with specific properties, for example, good guiding accuracy or robust control in systems with high uncertainties.

With a prepared ACS some mission scenarios will be demonstrated that are based on real satellite missions or standard maneuvers that are generally performed by three-axes stabilized spacecrafts.

6.1 Attitude determination

The attitude determination of a spacecraft is responsible for a reliable measurement of the attitude in space. This is often done with multiple sensors that verify each other's correct operation or enhance the accuracy of the measurement. A common task is the sensor fusion, in which different types of sensors are used together to estimate the attitude. A widespread combination of sensors is the use of a star camera for the attitude and rate gyros for the angular velocity. The camera can calculate a very precise attitude but only with a low update frequency, whereas the gyros have a high bandwidth but no absolute reference. The sensor fusion merges both measurements and estimates the attitude based on the knowledge of the characteristics of each sensor.

At this point, no sensor is available for an absolute reference, except for the magnetometers which are not precise enough for the intended purpose of the ACS. The attitude will only be estimated with the help of the rate gyros and a zero position. After the start of an experiment, equation 2.48 is used to integrate the angular rates, yielding the attitude of the experimental satellite. The integration starts immediately when a simulation begins, when the pneumatic actuators are still deployed and hold the assembly station horizontally. This is the zero position. The assembly station is still able to rotate around the vertical axis, which has no safety implication because the maximal deflection angles of the platform are always the same, but it is more comfortable to have a complete reference attitude for monitoring an experiment. A laser pointer attached to the lower side of the assembly station is aligned with a marked position on the wall to assure a similar starting attitude for every simulation. Refer to appendix A for the depicted reference frames and their attitude towards each other.

6.1.1 Filtering the Earth's angular rate from the rate gyros measurements

The reference system for the air bearing table is defined as an inertial frame that is fixed in the laboratory. The fiber optic rate gyros measure the Earth's rotation depending on their orientation towards the Earth's axis. Integrating the raw angular rates to obtain the attitude would cause a drift of the reference frame, hence the influence of the angular rate of the Earth must be removed. This also shows that defining the laboratory frame as inertial can only be an approximation.

For a period of $50\,\mathrm{s}$ at the beginning of each experiment, the angular rates are recorded. Several tests showed that this measuring period suffices for obtaining a good angular rate vector based on the noisy signals at the lower end of the gyros' sensor resolution. During the measurement the inertial frame is equal to the body fixed frame of the experimental satellite, which is held in place by the pneumatic actuators. The resulting angular rate $\underline{\omega}^i_{e,i}$, between the inertial and an Earth reference frame stays constant, but in body fixed coordinates it will change. Following this, the angular rate between the body fixed frame and the inertial (laboratory) system is

$$\underline{\omega}^b_{i,b} = \underline{\omega}^b_{e,b} - \underline{\underline{T}}^b_i \underline{\omega}^i_{e,i}\,. \tag{6.1}$$

The transformation matrix $\underline{\underline{T}}^b_i$ changes with the rotation of the experimental satellite. The attitude is the integral of the angular rates $\underline{\omega}^b_{i,b}$ over time, resulting in the quaternion $\mathbf{q}^b_i$ with the differential equation 2.48. The quaternion is either converted to the transformation matrix (cf. 2.49), or the rotation is computed solely by quaternion algebra.

6.1.2 Improving the angular rate measurement with a Kalman filter

Early tests with the fiber optic rate gyros showed significant noise on the sensor signal compared to the small angular rates of typical satellite maneuvers. A simple sliding averaging algorithm for obtaining a clearer signal to use in an attitude controller reacted too slowly to the changing angular rate. The properties of a Kalman filter promised a good noise suppression with better dynamic capabilities that justified the extra work for setting up and tuning the filter.

A Kalman filter is a state estimator that filters a sensor signal based on information about the sensor and a mathematical model of the process of which the sensor measures a certain property [9]. It compares the measurement with the predicted state of the process and gives an estimate for correct value. The filter is able to suppress large parts of normally distributed noise on the rate gyros' signals by estimating the state of the experimental satellite.

An implementation of an *extended Kalman filter* is used that works with a discrete and linearized model of the non-linear system dynamics of a spacecraft. The non-linear system is of the form

$$\dot{\underline{x}} = f(\underline{x}, \underline{u}, \underline{w})\,, \tag{6.2}$$

with $\underline{x}$ the state vector, $\underline{u}$ the control signal, and $\underline{w}$ the process noise. The measurement of the sensor is described as

$$\underline{z} = h(\underline{x}, \underline{v}) \,, \tag{6.3}$$

it depends on the actual system state and the noise $\underline{v}$ on the sensor signal. For the uses in the extended KALMAN filter, the state space defined by equations 6.2 and 6.3 must be linearized to the following form:

$$\begin{aligned} \dot{\underline{x}} &\approx \left.\frac{\partial f(\underline{x}, \underline{u}, \underline{0})}{\partial \underline{x}}\right|_{\underline{x}=\hat{\underline{x}}_-} \underline{x} = \underline{\underline{F}}\,\underline{x} \\ \underline{z} &\approx \left.\frac{\partial h(\underline{x}, \underline{0})}{\partial \underline{x}}\right|_{\underline{x}=\hat{\underline{x}}_-} \underline{x} = \underline{\underline{H}}\,\underline{x} \end{aligned} \tag{6.4}$$

As the operation point for the linearization the KALMAN filter uses the last estimated state vector $\hat{\underline{x}}_-$, implying that the matrices $\underline{\underline{F}}$ and $\underline{\underline{H}}$ change over time. Both matrices are necessary for the filter; in addition, four parameter matrices and the inital condition for the state vector are important to fine tune a KALMAN filter [10]:

$\underline{\underline{P}}_0$ The initial condition for the *state covariance matrix*, that will change during the use of the filter. This mainly influences the transient response of the filtered signal and therefore correlates to the time that has to pass before the filtered signal can be trusted.

$\underline{x}_0$ The initial condition of the state vector; the uncertainties in this condition must correlate to the initial condition for the state covariance matrix.

$\underline{\underline{Q}}$ The *system noise covariance matrix* is used to tell the filter how well the model represents the real process, and, in combination with the *measurement noise covariance matrix* $\underline{\underline{R}}$, describes which signal is more trustworthy: the modeled process or the measurements.

$\underline{\underline{R}}$ The *measurement noise covariance matrix* describes the noise of a measurement signal and relates to real physical properties of a sensor, which may or may not be found in its data sheet.

$\underline{\underline{G}}$ The *state noise coupling matrix* couples the noise of different elements of the state vector towards each other, for example, if one state is the integral of another state, their noise distributions correlate to each other. With the knowledge of these correlations, the filter can be tuned more accurately.

These four matrices are all used for statistically modeling the system and the measurement process. The covariances describe the spread of sensor signals or inaccuracies in the mathematical models. At the beginning of tuning a filter, most of the matrices can be set to an identity matrix or a multiple of it, but if, for example, the noise characteristics of a sensor are known, it should be used to reduce the numbers of parameters that have to be tuned.

State estimator for a constant angular velocity

The first filter for the angular rates of a fiber optic gyro will use a very simple model for the linear system inside the KALMAN filter. The angular velocity of the experimental satellite is chosen as the state vector

$$\underline{x} = \underline{\omega}_{i,b}^{b} \,. \tag{6.5}$$

This angular rate is estimated by the filter through evaluating the three sensor signals and the state space of a linear differential model. The derivative of the state vector is the angular acceleration that is assumed to be zero in all spatial directions. Therefore, the function f does not depend on the state $\underline{x}$, a control signal $\underline{u}$, or the process noise $\underline{w}$, where the latter is assumed to have an average of zero.

$$\dot{\underline{x}} = f(\underline{0}, \underline{0}, \underline{0}) = \dot{\underline{\omega}}_{i,b}^{b} = \underline{0} \tag{6.6}$$

The measurement function h is a direct mapping of the three gyro signals to the body fixed angular rates of the model.

$$\underline{z} = h(\underline{\omega}, \underline{0}) = \begin{bmatrix} \omega_{x\text{gyro}} \\ \omega_{y\text{gyro}} \\ \omega_{z\text{gyro}} \end{bmatrix}_{i,b}^{b} = \underline{\omega}_{i,b\text{gyro}}^{b} = \underline{\omega}_{i,b}^{b} \tag{6.7}$$

Even though this model is already linear, both functions from equations 6.6 and 6.7 are linearized as instructed in equation 6.4:

$$\begin{aligned} \underline{\underline{F}} &= \frac{\partial f}{\partial \underline{x}} = \frac{\partial \dot{\underline{\omega}}_{i,b}^{b}}{\partial \underline{\omega}_{i,b}^{b}} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \\ \underline{\underline{H}} &= \frac{\partial h}{\partial \underline{x}} = \frac{\partial \underline{\omega}_{i,b\text{gyro}}^{b}}{\partial \underline{\omega}_{i,b}^{b}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{aligned} \tag{6.8}$$

Finally, the parameter matrices must be defined that tune the dynamic behavior of the KALMAN filter:

$$\begin{aligned} \underline{\underline{G}} &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad \underline{\underline{Q}} = \begin{bmatrix} \sigma_Q^2 & 0 & 0 \\ 0 & \sigma_Q^2 & 0 \\ 0 & 0 & \sigma_Q^2 \end{bmatrix} \qquad \underline{\underline{R}} = \begin{bmatrix} \sigma_{\text{gyro}}^2 & 0 & 0 \\ 0 & \sigma_{\text{gyro}}^2 & 0 \\ 0 & 0 & \sigma_{\text{gyro}}^2 \end{bmatrix} \\ \underline{\underline{P}}_0 &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad \underline{x}_0 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \end{aligned} \tag{6.9}$$

The noise coupling matrix $\underline{\underline{G}}$ is set to an identity matrix, because the noise distributions of the gyros are not correlated to each other. For the main diagonal of the system noise matrix $\underline{\underline{Q}}$, one variable suffices for tuning the filter, because the system and the sensors are

symmetric in all dimensions. This variable can be tuned with regard to the measurement matrix $\underline{\underline{R}}$ that is well defined by the known characterization of the sensor's noise with equation 3.4:

$$\sigma^2_{\text{gyro}} = \eta^2_{\text{gyro}} f_{\text{sampling}} = \left(\frac{0.15^\circ}{\sqrt{3600\,\text{s}}} \frac{\pi}{180^\circ} \right)^2 \cdot 100\,\text{Hz} \approx 1.9 \cdot 10^{-7}\,\text{rad}^2/\text{s}^2 \tag{6.10}$$

If the covariance σ^2_Q, the square of the standard deviation, is smaller than σ^2_{gyro}, the filter will trust the mathematical model more than the measurements of the μFORS gyros; if σ^2_Q is chosen very high, this indicates that the model should not be trusted as much, compared to the good sensor quality. $\underline{\underline{P}}_0$ and $\underline{x}_0$ are the initial conditions for the covariance matrix of the state and the state itself, respectively.

State estimator for a variable angular velocity of a satellite

The second Kalman filter design uses a more sophisticated model that the measured angular rates are compared to. Again, the basic model is the dynamics of a satellite with reaction wheels for attitude control from equations 2.42 and 5.7. The reference system is the laboratory frame i that is assumed to be inertial.

The state vector is the same as for the previous design: $\underline{x} = \underline{\omega}^b_{i,b}$. This time, the state space is a function of the angular rates and the control input $\underline{u}$ of the reaction wheels' angular momentum and its derivative.

$$\dot{\underline{x}} = f(\underline{x}, \underline{u}, \underline{0}) = f\left(\underline{\omega}^b_{i,b}, \begin{bmatrix} \underline{h}^b_{b,\mathbf{W}} \\ \dot{\underline{h}}^b_{b,\mathbf{W}} \end{bmatrix}, \underline{0} \right) = \dot{\underline{\omega}}^b_{i,b} \tag{6.11}$$

The noise is assumed to be normally distributed with an average of zero. Before linearizing the dynamic model of the satellite

$$\dot{\underline{\omega}}^b_{i,b} = {\underline{\underline{I}}^b_{\mathbf{B}}}^{-1} \left(-\underline{\omega}^b_{i,b} \times \left(\underline{\underline{I}}^b_{\mathbf{B}} \underline{\omega}^b_{i,b} \right) - \underline{\omega}^b_{i,b} \times \underline{h}^b_{b,\mathbf{W}} - \dot{\underline{h}}^b_{b,\mathbf{W}} \right), \tag{6.12}$$

the cross product must be swapped with a matrix multiplication by mapping the first factor of a cross product to a skew-symmetric matrix of size 3×3. For example, the cross product of two vectors $\underline{a}$ and $\underline{b}$ is the same as a matrix multiplication of the following type:

$$\underline{a} \times \underline{b} = \begin{bmatrix} 0 & -a_z & a_y \\ a_z & 0 & -a_x \\ -a_y & a_x & 0 \end{bmatrix} \underline{b} = \underline{\underline{a}}\,\underline{b} \tag{6.13}$$

A second line under the symbol is used to indicate the skew-symmetric form of a vector, which is consistent with the nomenclature for a matrix. As for the cross product, the same rule applies for a change in the order if using the skew-symmetric multiplication: $\underline{\underline{a}}\,\underline{b} = -\underline{\underline{b}}\,\underline{a}$.

Using only matrix multiplications, the satellite's dynamic is

$$\dot{\underline{\omega}}^b_{i,b} = {\underline{\underline{I}}^b_{\mathbf{B}}}^{-1} \left(-\underline{\underline{\omega}}^b_{i,b} \left(\underline{\underline{I}}^b_{\mathbf{B}} \underline{\omega}^b_{i,b} \right) - \underline{\underline{\omega}}^b_{i,b} \underline{h}^b_{b,\mathbf{W}} - \dot{\underline{h}}^b_{b,\mathbf{W}} \right). \tag{6.14}$$

Differentiating the function f with respect to the system state is now a straightforward task.

$$\begin{aligned}\underline{\underline{F}} = \frac{\partial f}{\partial \underline{x}} &= \frac{\partial_b \dot{\underline{\omega}}^b_{i,b}}{\partial_b \underline{\omega}^b_{i,b}} \\ &= \underline{\underline{I}}^{b\,-1}_{\mathbf{B}} \left(\frac{\partial_b \left(-\underline{\underline{\omega}}^b_{i,b} \left(\underline{\underline{I}}^b_{\mathbf{B}} \underline{\omega}^b_{i,b} \right) \right)}{\partial_b \underline{\omega}^b_{i,b}} + \frac{\partial_b \left(-\underline{\underline{\omega}}^b_{i,b} \underline{h}^b_{b,\mathbf{W}} \right)}{\partial_b \underline{\omega}^b_{i,b}} + \frac{\partial_b \left(\dot{\underline{h}}^b_{b,\mathbf{W}} \right)}{\partial_b \underline{\omega}^b_{i,b}} \right) \qquad (6.15) \\ &= \underline{\underline{I}}^{b\,-1}_{\mathbf{B}} \left(-\underline{\underline{\omega}}^b_{i,b} \left(\underline{\underline{I}}^b_{\mathbf{B}} \frac{\partial_b \underline{\omega}^b_{i,b}}{\partial_b \underline{\omega}^b_{i,b}} \right) + \underline{\left[\underline{\underline{I}}^b_{\mathbf{B}} \underline{\omega}^b_{i,b} \right]} \frac{\partial_b \underline{\omega}^b_{i,b}}{\partial_b \underline{\omega}^b_{i,b}} + \underline{\underline{h}}^b_{b,\mathbf{W}} \frac{\partial_b \underline{\omega}^b_{i,b}}{\partial_b \underline{\omega}^b_{i,b}} \right) \qquad (6.16)\end{aligned}$$

The first step is to split the differentiation into three parts. The product rule is applied to the first part, resulting in two summands. The second summand of the product rule is lead by a term in squared brackets that is underlined. This is the skew-symmetric form of the resulting vector of the multiplication inside the brackets. The second term of the differentiation is merely reordered, and the third part vanishes because it does not depend on the angular rate $\underline{\omega}^b_{i,b}$.

The term $\frac{\partial_b \underline{\omega}^b_{i,b}}{\partial_b \underline{\omega}^b_{i,b}}$ can be omitted because it is the identity matrix. This leads to the final expression for the linearization matrix $\underline{\underline{F}}$.

$$\underline{\underline{F}} = \underline{\underline{I}}^{b\,-1}_{\mathbf{B}} \left(-\underline{\underline{\omega}}^b_{i,b} \underline{\underline{I}}^b_{\mathbf{B}} + \underline{\left[\underline{\underline{I}}^b_{\mathbf{B}} \underline{\omega}^b_{i,b} \right]} + \underline{\underline{h}}^b_{b,\mathbf{W}} \right) \qquad (6.17)$$

All other matrices are set up the same way as for the KALMAN filter in the last section, including the tunable parameter σ_Q^2.

Filter comparison and tests

Comparing and assessing the two KALMAN filters requires a simulation of the satellite dynamics. A simulation has the advantage that it is repeatable and faster than actually using the air bearing table. Additionally, an experiment would be much more complicated to evaluate, because the correct value of the angular rate is not known, as no external reference for the attitude is available.

The simulation uses the same mathematical description of the dynamics of the satellite as used for the design of the second filter and for the measurement of the moment of inertia in section 5.2. For a detailed simulation of the utilized attitude hardware, their accuracy and noise behavior must be reproduced.

The variance for the optic gyros was already used for tuning the filter and can be used to generate the noise on the angular rate signals of the simulation. The user manual of the reaction wheels [8] and initial operation test protocols from the manufacturer describe the accuracy of the angular rate measurement of the flywheel with a standard deviation of $\sigma_{\omega\text{flywheel}} = 0.26\,\text{rpm}$, and the deviation towards the commanded torque with $\sigma_T = 8.54\,\text{mNm}$.

This information is used to simulate the attitude hardware of the experimental satellite. The noise magnitude of the torque was modified to have less influence if only a low torque is commanded, which is an actual characteristic of the utilized reaction wheels. The build simulation is only a limited model of the real satellite, but it should suffice for the purpose of comparing and tuning both implemented filters.

The test scenario is a slew maneuver with an accelerating and a decelerating phase and a total rotation angle of $30°$ around an arbitrarily chosen axis. The first diagram in Fig. 6.2 shows the resulting angular rates for each body fixed axis with the simulated measuring noise. The maneuver was simulated with different values of the only tuning parameter σ_Q for each KALMAN filter. The evaluation criterion for the filters is the average error $\bar{\epsilon}$ of the norm of the estimated angular rates towards the actual rates during the whole maneuver. Fig. 6.1 shows the result of the parameter study for three different cases. First, the simple KALMAN filter that models a constant angular velocity $\bar{\epsilon}_{\omega\,\mathrm{const}}$ and two different variants of the more complex system model. One time, the filter $\bar{\epsilon}_{\mathrm{exact}}$ is using the exact same moment of inertia as the simulation of the attitude dynamics, and the other time $\bar{\epsilon}_{\mathrm{coarse}}$, it is set to a value that was also used in the process of measuring the moment of inertia itself to determine the influence on the filter when using an inaccurate moment of inertia matrix. For this case, the values for the principal moments of inertia were chosen approximately $20\,\%$ too high, assuming an experimental satellite that is less agile than it actually is, to avoid stability problems during the initial operations. Additionally, the mean error $\bar{\epsilon}_{\mathrm{gyro}}$

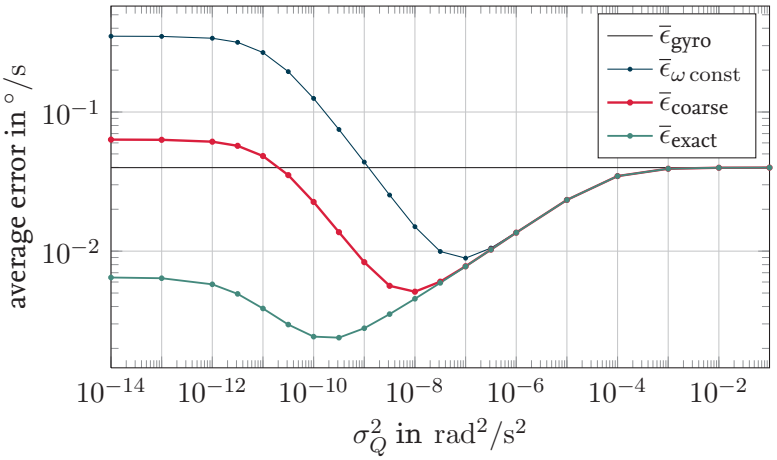


Figure 6.1: Signal quality of the filtered angular rates

of the unfiltered signal is shown in the graph. A smaller value of σ_Q^2 results in a filter that trusts the mathematical model more than the measurements of the gyros. For very small values all three filter implementations differ, and only if a good estimate for the moment of inertia is used, the modeled dynamics are accurate enough to improve the overall signal quality compared to the unfiltered measurements.

High values of σ_Q^2 result in a similar error for all filters, because the gyros' values gain more weight in the estimation process, and they are identical for all implementations.

All three cases have a minimum in their average error that significantly improves the measurement. A comparison of the signal quality of theses minima is given in the second diagram of Fig. 6.2 with the corresponding value for σ_Q^2. Even with a bad estimate for

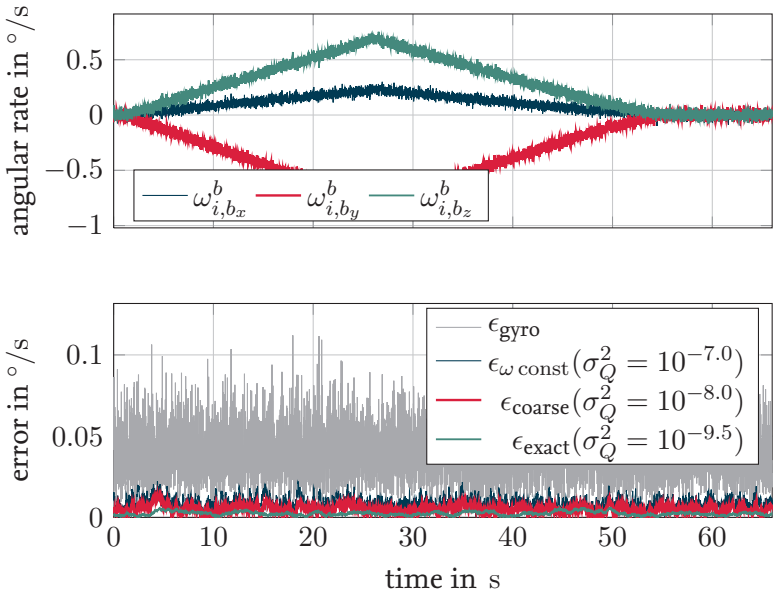


Figure 6.2: Maneuver profile and error towards the actual angular velocity of the different Kalman filters

the moment of inertia, the signal is smoother with an overall smaller error in the signal, and for good estimates, the noise is almost insignificant compared to the sensor values in contrast to the noise distribution of the unfiltered signal. If a filter including the satellite dynamics is implemented on the air bearing table, care must be taken because an off-centered c. m., or any other disturbance, would be compensated by the wheels, influencing the filter by assuming an accelerating motion that leads to wrong attitude information and a drift in the reference system if the angular rates are integrated over time.

Improvements of the attitude estimation are of course possible, but would require a deeper insight into the matter of Kalman filtering and modeling the attitude hardware more precisely. One idea would be to include the measurements of the reaction wheels' angular velocities and their derivatives in the filter for better results. In that case, the linearization in equation 6.17 would be more accurate, which should influence the overall performance of the filter. Additionally, different types of Kalman filter should be used, for example an *unscented Kalman filter* [9].

6.2 Attitude control

The attitude control is responsible for the reduction of the deviation between the measured attitude and a desired guidance attitude. The guidance attitude can be constant or

changing over time if, for example, a ground station has to be tracked from orbit. For this purpose a simple linear controller design is used that requires a linearized system model of the satellite. Contrary to the linearization approach for the attitude determination, the PID controller is only designed for a specific operation point that results in a simpler description of the system's dynamics. The disadvantage of this is that the controller can only compensate disturbances on the basis of the linearized model. For a better control quality the known gyroscopic couplings in the basic satellite model from equation 2.42 are calculated and used for an open-loop control in addition to the closed-loop control with the PID controller.

The state space of a linear system is commonly described by the following two equations [11, 12]:

$$\dot{\underline{x}} = \underline{\underline{A}}\,\underline{x} + \underline{\underline{B}}\,\underline{u} \tag{6.18}$$

$$\underline{y} = \underline{\underline{C}}\,\underline{x} + \underline{\underline{D}}\,\underline{u} \tag{6.19}$$

The change of the state vector $\dot{\underline{x}}$ depends on the state vector itself and the *control vector* $\underline{u}$. The dynamics of the linear system are described by the *state matrix* $\underline{\underline{A}}$ and the *input matrix* $\underline{\underline{B}}$, whereas the output of the system $\underline{y}$ is defind by an *output matrix* $\underline{\underline{C}}$ and a *feedthrough matrix* $\underline{\underline{D}}$. The output equation is just given for completeness, but for this purpose, the output is assumed to be equal to the state vector by choosing an identity matrix for $\underline{\underline{C}}$ and a zero matrix for $\underline{\underline{D}}$.

Two different controllers are designed for controlling of the experimental satellite on the air bearing table. A robust proportional-derivative (PD) controller, which even works with high uncertainties in the linear state space with respect to the real system, and a proportional-integral-derivative (PID) controller for a better guidance accuracy in the demonstration scenarios.

6.2.1 Linear dynamics for the PD controller

A linearized model of the satellite's dynamics is the basis to design a PD controller. The controller uses the deviation between a guidance trajectory and the measured satellite's state to compute the necessary torque to minimize the error. The definition of the control deviation for the angular rate is simply the difference between the guidance and actual rate $\underline{\omega}^b_{b,g}$, whereas the difference between the guidance attitude $\mathbf{q}^g_i$ and the measured attitude $\mathbf{q}^b_i$ must be computed via a quaternion multiplication [13].

$$\begin{aligned} \mathbf{q}^g_b &= \mathbf{q}^i_b \mathbf{q}^g_i \text{ , with } \mathbf{q}^i_b = \overline{\mathbf{q}}^b_i \\ \underline{\omega}^b_{b,g} &= \underline{\omega}^b_{i,g} - \underline{\omega}^b_{i,b} \end{aligned} \tag{6.20}$$

This deviation from the guidance is a zero offset, which allows the controller to operate only in the surrounding of zero. This has the advantage that the linearized state space only has to model a small part of the system's dynamics around an operation point, which is defined as

$$\mathbf{q}_0 = \begin{Bmatrix} 0 & 0 & 0 & 1 \end{Bmatrix}^T \text{ and } \underline{\omega}_0 = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}^T . \tag{6.21}$$

The aim of the controller is to maneuver the satellite to the desired rate $\underline{\omega}_{b,g}^{b} \rightarrow \underline{\omega}_0$ and attitude $\mathbf{q}_b^g \rightarrow \mathbf{q}_0$.

The state vector for the PD controller includes the attitude $\mathbf{q}_b^g$ and the angular rate $\underline{\omega}_{b,g}^{b}$ of the experimental satellite relative to the guidance. The changing attitude is described by equation 2.48 and the modeled angular acceleration from equation 2.42 and 6.12, respectively, is further simplified by neglecting the terms with gyroscopic couplings for the angular momentum of the satellite and the angular momentum of the saturated reaction wheels, which are taken into account by a feed forward control.

$$\underline{x} = \begin{bmatrix} \mathbf{q}_b^g \\ \underline{\omega}_{b,g}^{b} \end{bmatrix} \qquad \underline{\dot{x}} = \begin{bmatrix} \dot{\mathbf{q}}_b^g \\ \underline{\dot{\omega}}_{b,g}^{b} \end{bmatrix} = \begin{bmatrix} \frac{1}{2}\underline{\underline{\Omega}}_{b,g}^{b}\mathbf{q}_b^g \\ \underline{\underline{I}}_{\mathbf{B}}^{b\,-1}\left(-\underline{\dot{h}}_{b,\mathbf{W}}^{b}\right) \end{bmatrix} \tag{6.22}$$

From the definition in equation 2.45 of a quaternion for rotations it is known that the fourth element is the cosine of the half rotation angle, which can be approximated as one for small angles. For this reason, only the first three elements of the attitude quaternion are used in the state vector $\underline{x}$, because almost no new control information can be extracted from the fourth that is not already part of the first three elements.

With the above preparations it is possible to linearize the model from equation 6.22 and define the state matrix:

$$\underline{\underline{A}}_{PD} = \left.\frac{\partial \underline{\dot{x}}}{\partial \underline{x}}\right|_{\mathbf{q}_0,\underline{\omega}_0} = \left.\begin{bmatrix} \frac{\partial \dot{\mathbf{q}}_b^g}{\partial \mathbf{q}_b^g} & \frac{\partial \dot{\mathbf{q}}_b^g}{\partial \underline{\omega}_{b,g}^{b}} \\ \frac{\partial \underline{\dot{\omega}}_{b,g}^{b}}{\partial \mathbf{q}_b^g} & \frac{\partial \underline{\dot{\omega}}_{b,g}^{b}}{\partial \underline{\omega}_{b,g}^{b}} \end{bmatrix}\right|_{\mathbf{q}_0,\underline{\omega}_0} = \begin{bmatrix} 0 & 0 & 0 & 0.5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.5 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{6.23}$$

Most derivatives are zero because of the simple model and of the conveniently chosen operation point of zero. Only the dependency of the attitude change towards the angular rate results in $\frac{1}{2}$ on the diagonal. The input matrix is completely defined by the negative and inverted moment of inertia of the satellite:

$$\underline{\underline{B}}_{PD} = \left.\frac{\partial \underline{\dot{x}}}{\partial \underline{u}}\right|_{\mathbf{q}_0,\underline{\omega}_0} = \left.\begin{bmatrix} \frac{\partial \dot{\mathbf{q}}_b^g}{\partial \underline{\dot{h}}_{b,\mathbf{W}}^{b}} \\ \frac{\partial \underline{\dot{\omega}}_{b,g}^{b}}{\partial \underline{\dot{h}}_{b,\mathbf{W}}^{b}} \end{bmatrix}\right|_{\mathbf{q}_0,\underline{\omega}_0} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ -\underline{\underline{I}}_{\mathbf{B}}^{b\,-1} \end{bmatrix} \tag{6.24}$$

The output matrix $\underline{\underline{C}}$ is an identity matrix of size 6×6 and the feedthrough matrix $\underline{\underline{D}}$ is zero. This completes the description of the linear dynamics for the PD controller, but before the actual controller is designed, the dynamics must be adapted for the PID controller.

6.2.2 Linear dynamics for the PID controller

The state vector for the linear dynamics must be extended with a third element that describes the integral controller error towards the trajectory. The integral deviation is

$$\underline{e} = \int_{t_1}^{t_2} \mathbf{q}_b^g \mathrm{d}t\,. \tag{6.25}$$

The integration is done from the time t_1 when the controller is enabled to the present time t_2. Again, just the first three elements of the integral deviation are used for the dynamics. This concludes the state vector and its derivative for the PID controller as

$$\underline{x} = \begin{bmatrix} \mathbf{q}_b^g \\ \underline{\omega}_{b,g}^b \\ \underline{e} \end{bmatrix} \qquad \text{and} \qquad \dot{\underline{x}} = \begin{bmatrix} \dot{\mathbf{q}}_b^g \\ \dot{\underline{\omega}}_{b,g}^b \\ \dot{\underline{e}} \end{bmatrix} = \begin{bmatrix} \frac{1}{2}\underline{\underline{\Omega}}_{b,g}^b \mathbf{q}_b^g \\ \underline{\underline{I}}_{\mathbf{B}}^{b\,-1} \left(-\dot{\underline{h}}_{b,\mathbf{W}}^b \right) \\ \mathbf{q}_b^g \end{bmatrix}, \tag{6.26}$$

with the attitude deviation $\mathbf{q}_b^g$ as the derivative of the integral error $\underline{e}$. The operation point for the linearization is identical to $\mathbf{q}_0$ and ω_0 as given in equation 6.21. This concludes the state matrix $\underline{\underline{A}}$ and the input matrix $\underline{\underline{B}}$ for the PID controller.

$$\underline{\underline{A}}_{PID} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0.5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.5 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \qquad \underline{\underline{B}}_{PID} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ -\underline{\underline{I}}_{\mathbf{B}}^{b\,-1} \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{6.27}$$

6.2.3 Linear quadratic regulator

The linear dynamics from the two sections above are used to calculate the gains for the controller. The employed method for this is called LQR, which is a widespread approach for robust controller design [14, 12]. It is based on the optimization of the linear dynamics' response for a fast decrease of control deviation or system state $\underline{x}$ and control signal $\underline{u}$. The cost function for this problem is commonly stated as:

$$J = \int_0^\infty \left(\underline{x}^T \underline{\underline{Q}}\, \underline{x} + \underline{u}^T \underline{\underline{R}}\, \underline{u} \right) \mathrm{d}t \tag{6.28}$$

The longer it takes to reach a steady state of the system, the greater the cost value J is. The two matrices $\underline{\underline{Q}}$ and $\underline{\underline{R}}$ must be defined in a way to tune the optimization process in order to obtain a controller with the desired properties. The matrix $\underline{\underline{Q}}$ penalizes the error of the state vector towards the operation point, and $\underline{\underline{R}}$ is used to value the usage of the actuators, which otherwise would accumulate unbounded throughout the optimization process. Usually, only the main diagonals are used to tune the controller. Each diagonal element penalizes the corresponding element of the state vector or the control vector:

$$\underline{\underline{Q}}_{PD} = \mathrm{diag}(\underline{Q}_{\mathbf{q}}, \underline{Q}_{\omega}) \qquad \underline{\underline{Q}}_{PID} = \mathrm{diag}(\underline{Q}_{\mathbf{q}}, \underline{Q}_{\omega}, \underline{Q}_{e}) \tag{6.29}$$

$$\underline{\underline{R}}_{PD} = \underline{\underline{R}}_{PID} = \mathrm{diag}(\underline{R}_{w}) \tag{6.30}$$

The subscripts indicate the corresponding state vector elements, and the vector $\underline{R}_w$ penalizes the the usage of the reaction wheels.

For the computation of the controller gain $\underline{\underline{K}}$, a software implementation from the Matlab™ development environment is used. The function call for the automatic optimization for both controllers is

$$\underline{\underline{K}}_{PD} = \mathrm{lqrd}(\underline{\underline{A}}_{PD}, \underline{\underline{B}}_{PD}, \underline{\underline{Q}}_{PD}, \underline{\underline{R}}_{PD}, 0.5)\,, \tag{6.31}$$

$$\underline{\underline{K}}_{PID} = \mathrm{lqrd}(\underline{\underline{A}}_{PID}, \underline{\underline{B}}_{PID}, \underline{\underline{Q}}_{PID}, \underline{\underline{R}}_{PID}, 0.5)\,. \tag{6.32}$$

The last parameter of the function call is the discrete sampling time for the controller, which is defined by the reaction wheels' maximal command frequency of 2 Hz (see section 3.2.2). Refer to the appropriate literature for the complete description of the optimization algorithm.

The general control law for the computed controller gains is

$$\underline{u} = \underline{\underline{K}}\,\underline{x}\,, \tag{6.33}$$

where $\underline{u}$ equals the torque for the acceleration of the reaction wheels. The uncommon definition of this control law, which uses the flywheel's acceleration, not the satellite's acceleration, results from the definition of the control torque as $\dot{\underline{h}}^{\,b}_{b,\mathbf{W}}$ and not its negated value.

The diagram of the process controlling the experimental satellite is depicted in Fig. 6.3 with the control loop highlighted in blue. The torque from the PD/PID controller is re-

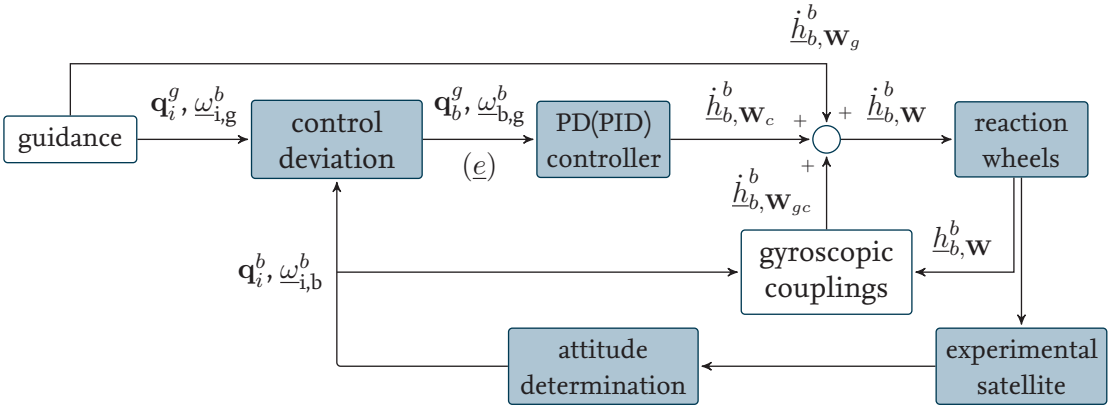


Figure 6.3: Control diagram of the experimental satellite's ACS

defined as $\dot{\underline{h}}^{\,b}_{b,\mathbf{W}_c}$, because the summation with additional torques for the feed forward control form the overall torque commanded to the reaction wheels. The blocks named *gyroscopic couplings* and *guidance* are descried in the following section.

6.2.4 Feed forward control

Additional knowledge about a closed-loop controlled system that is not covered by the dynamics on which the controller is based can be used to improve the control quality by

using a *feed forward control*, or, by planning a maneuver ahead, an expected control signal can be generated along the maneuver trajectory for a better guidance accuracy.

Gyroscopic couplings

The linear controllers were only designed for the simplified linear dynamics of the satellite's motion (cf. equations 6.22 and 6.26). The omitted terms act like disturbances on the system, which the controller has to compensate for. By explicitly calculating the gyroscopic couplings $\dot{\underline{h}}^{b}_{b,\mathbf{W}_{\text{gc}}}$, and adding them to the control signal, the controller only has to act on the deviations resulting from other disturbances.

$$\dot{\underline{h}}^{b}_{b,\mathbf{W}_{gc}} = -\underline{\omega}^{b}_{i,b} \times \left(\underline{\underline{I}}^{b}_{\mathbf{B}} \underline{\omega}^{b}_{i,b}\right) - \underline{\omega}^{b}_{i,b} \times \underline{h}^{b}_{b,\mathbf{W}} \tag{6.34}$$

This control needs to know the angular momentum of the reaction wheels and the moment of inertia tensor of the experimental satellite. The second coupling can be computed from the housekeeping data of the wheels, but if, say, the moment of inertia is not known with sufficient accuracy, this control could render the overall control loop from Fig. 6.3 instable, whereas small uncertainties in the measurement of these additional quantities will be compensated for by the closed-loop controller [12].

Guidance

The linear controllers are only capable of producing good results in the surrounding of the operation point. If the control deviations are too high, the non-linearities gain influence, which is the motivation for using a guidance trajectory for open-loop control. The trajectory is defined between two attitudes that stay constant during the slew maneuver. It starts with the starting attitude $\mathbf{q}^{s}_{i}$ and ends with the target attitude $\mathbf{q}^{t}_{i}$. The following equation yields a difference quaternion that specifies the rotation the satellite has to carry out to complete the maneuver:

$$\mathbf{q}^{t}_{s} = \mathbf{q}^{i}_{s}\mathbf{q}^{t}_{i} = \overline{\mathbf{q}}^{s}_{i}\mathbf{q}^{t}_{i} \tag{6.35}$$

From this quaternion, two pieces of information can be extracted according to equation 2.45: the total rotation angle ε_t and the rotation axis $\underline{n}$.

$$\mathbf{q}^{t}_{s} \rightarrow \varepsilon_t, \underline{n} \tag{6.36}$$

For this rotation a *slope limiter* is used to define the trajectory for a scalar rotation angle ε_t [15]. The output of this slope limiter is a time-dependent control torque $\dot{h}^{b}_{b,\mathbf{W}_g}$ that is based on restrictions concerning the maximal allowed torque $\dot{h}^{b}_{b,\mathbf{W}_{\max}}$ and angular rate $\omega^{b}_{i,b_{\max}}$, and also uses the satellite's moment of inertia as a parameter. The first diagram in Fig. 6.4 shows the course of an exemplary control torque.

The angular acceleration is defined using the known one-dimensional moment of inertia around the rotation axis $\underline{n}$:

$$\dot{\omega}^{b}_{i,b} = -\frac{\dot{h}^{b}_{b,\mathbf{W}_g}(\underline{\underline{I}}^{b}_{\mathbf{B}}, \omega^{b}_{i,b_{\max}}, \dot{h}^{b}_{b,\mathbf{W}_{\max}}, t)}{\underline{n}^{T}\underline{\underline{I}}^{b}_{\mathbf{B}}\underline{n}} \tag{6.37}$$

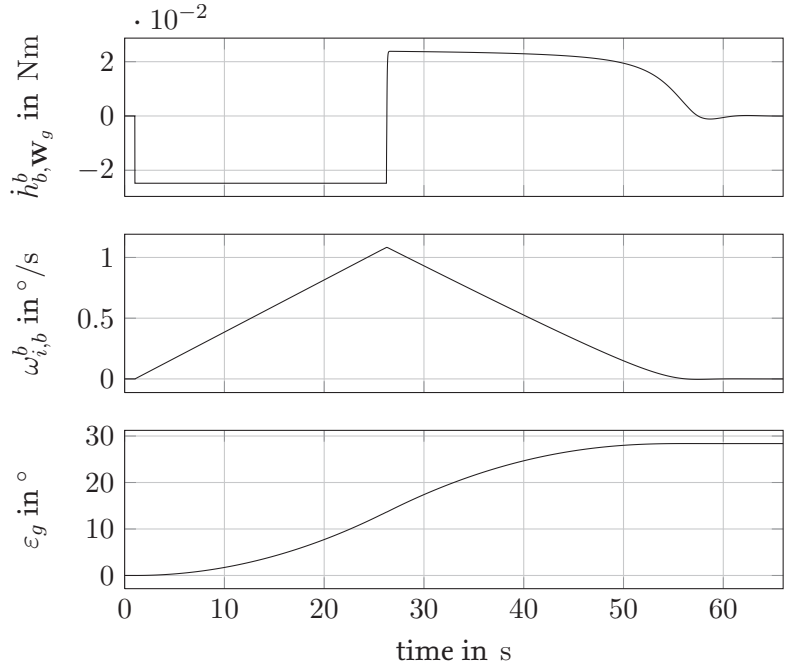


Figure 6.4: Guidance trajectory of a slew maneuver

Integrating the acceleration twice from the beginning of the maneuver t_1 throughout the course of the maneuver yields a time varying guidance angle ε_g and angular rate $\omega^b_{i,b}$. This is shown in the second and third diagram in Fig. 6.4 for the exemplary control torque.

$$\varepsilon_g(t_x) = \int_{t_1}^{t_x} \omega^b_{i,b} \mathrm{d}t = \int_{t_1}^{t_x} \int_{t_1}^{t_x} \dot{\omega}^b_{i,b} \mathrm{d}t \text{ , with } \varepsilon_g(t_1) = \omega^b_{i,b}(t_1) = 0 \tag{6.38}$$

The defined one-dimensional slew maneuver must be transformed to the three-dimensional body frame. This is done by reversing the above extraction from the quaternion to obtain the time-varying difference quaternion $\mathbf{q}^g_r$:

$$\mathbf{q}^g_s \leftarrow \varepsilon_g, \underline{n} \tag{6.39}$$

The guidance attitude $\mathbf{q}^g_i$ is the product of the difference quaternion and the reference quaternion, whereas the scalar guidance rate and torque are multiplied by the rotation axis:

$$\begin{aligned} \mathbf{q}^g_i &= \mathbf{q}^s_i \mathbf{q}^g_s \\ \underline{\omega}^b_{i,b} &= \omega^b_{i,b} \underline{n} \\ \underline{\dot{h}}^b_{b,\mathbf{W}_g} &= \dot{h}^b_{b,\mathbf{W}_g} \underline{n} \end{aligned} \tag{6.40}$$

These trajectories can be fed into the guidance block in the control diagram 6.3 to support the closed-loop control for a better guiding accuracy.

6.2.5 Controller tests

The tuning matrices for the robust PD controller are chosen for a good guidance accuracy without too much suppression of a high activity of the actuators. This can result in small oscillations around the operation point. Deviations in the angular rate and attitude are penalized more severely than the use of the actuators: $\underline{Q}_{\mathbf{q}}, \underline{Q}_{\omega} > \underline{Q}_{w}$.

$$\underline{\underline{Q}}_{PD} = \mathrm{diag}(\underline{Q}_{\mathbf{q}}, \underline{Q}_{\omega}) \qquad = \mathrm{diag}(10, 10, 10, 10, 10, 10) \tag{6.41}$$

$$\underline{\underline{R}}_{PD} = \mathrm{diag}(\underline{R}_{w}) \qquad = \mathrm{diag}(1, 1, 1) \tag{6.42}$$

In multiple experiments, this approach has proven to result in a controller that reacts quickly and robustly if the deviations from the guidance are high, or even if a step occurs in the target quaternion. For example, if an experiment is aborted in the middle of a maneuver, this controller safely returns the assembly station back to its home position.

The priorities for the PID controller are chosen for a better guidance accuracy with a more cautious control signal in order to prevent most of the oscillations around the trajectory.

$$\underline{\underline{Q}}_{PID} = \mathrm{diag}(\underline{Q}_{\mathbf{q}}, \underline{Q}_{\omega}, \underline{Q}_{e}) \qquad = \mathrm{diag}(10, 10, 10, 100, 100, 100, 100, 100, 100) \tag{6.43}$$

$$\underline{\underline{R}}_{PID} = \mathrm{diag}(\underline{R}_{w}) \qquad = \mathrm{diag}(10000, 10000, 10000) \tag{6.44}$$

This design shows its advantages especially in the scenario of aligning the satellite with a ground station. Even though it is possible that the PID controller becomes instable, it is still robust in a lot of situations, for example, if one of the beams of the assembly station slightly touches the support stator. For the case of an instable reaction of the ACS, it is advisable to take precautions in order to prevent damage to the hardware, like switching to the PD controller.

A short guidance trajectory was used to compare the controllers, and for the positive impact of the feed forward control on the guidance accuracy. The trajectory is computed between three different attitude quaternions, using the guidance from section 6.2.4. The first diagram in Fig. 6.5 shows the profile of the deflection angles for the maneuvers. First, the satellite rotates around the x axis. Afterwads, it rotates to an attitude with only a y deflection, finally followed by a z rotation before the home position is targeted again. All the experiments were realized in the FACE with reaction wheels that were accelerated to $3000\,\mathrm{rpm}$ prior to the guidance tests in order to show the impact of the gyroscopic couplings.

The first test was executed with an active forward control and controllers that were optimized for a coarsely estimated moment of inertia. By comparison, the PD controller was more capable of following the guidance trajectory than the PID controller. The maximal deviation from the guidance $\varepsilon^{b}_{g,b}$ was twice as high as the the PD controller's, similar to the average error towards the trajectory $\bar{\varepsilon}$ given in the legend of the diagram. Between $250\,\mathrm{s}$ and $275\,\mathrm{s}$ after the experiment was started, the above mentioned oscillation of the PD controller is visible, whereas the PID controller follows the trajectory with a smoother shape.

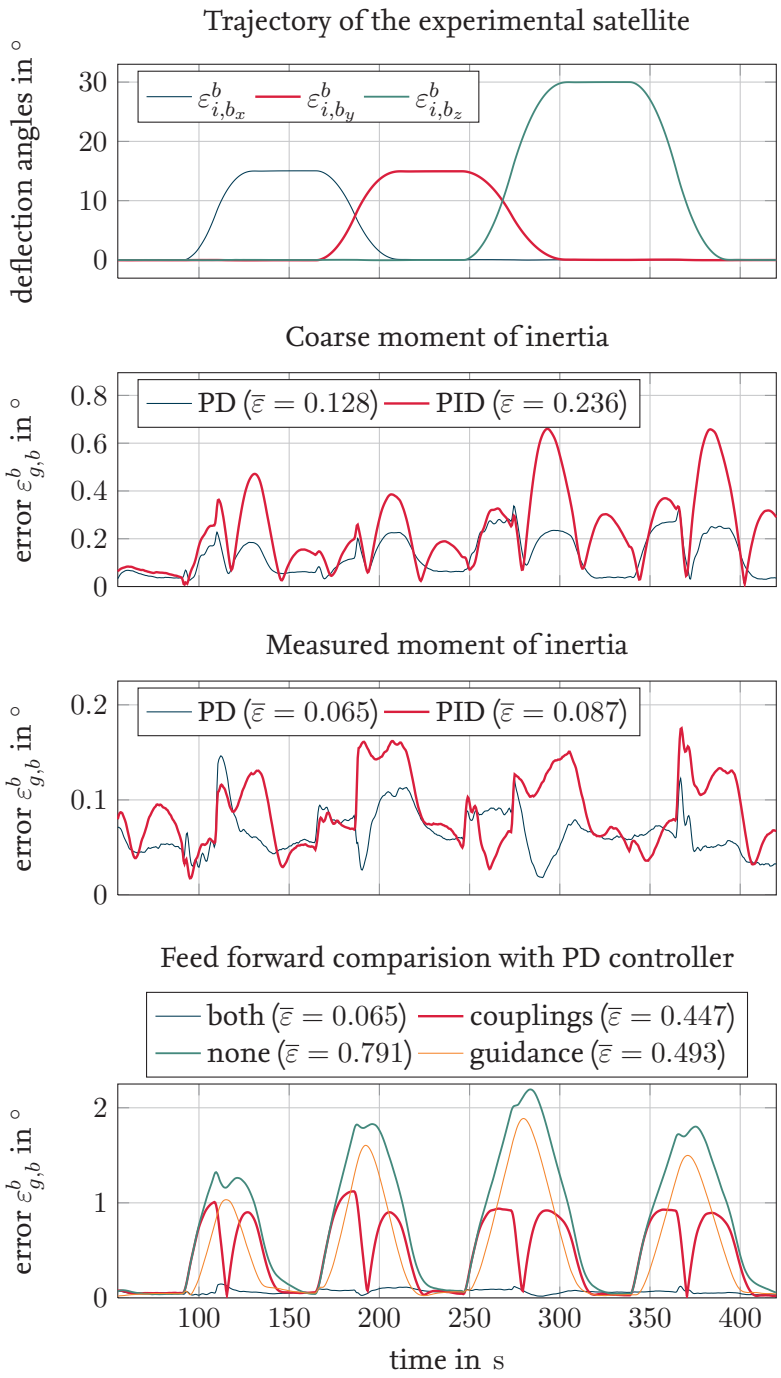


Figure 6.5: Evaluation of the ACS with different controlling strategies

The third diagram in Fig. 6.5 shows the comparison of the two controllers that were designed using the measured moment of inertia from section 5.2. The PD controller has a

better average deviation from the guidance, but compared to the difference between the two controllers from the last test it is less noticeable. The average guidance accuracy is smaller than one tenth of a degree, which is a good result, especially considering the step of the torque at the beginning of each slew maneuver (see the first diagram of Fig. 6.4), which induces an oscillation into the closed-loop controlled system. The overall possible accuracy strongly depends on the angular rates during the maneuver; when choosing smaller values for the rate and acceleration limiting parameters of the guidance (cf. equation 6.37) better results can be expected.

The last controller test was made to show the effectiveness of the feed forward control. The PD controller was used to follow the guidance under different circumstances. The first run was once more executed with both feed forward implementations; the second run was completely controlled without a forward control. The final two runs each used only one of the two forward controls. Not using the guidance feed forward controll in this case means that the torque $\underline{\dot{h}}^{b}_{b,\mathbf{W}_g}$ in the control diagram (6.3) is set to zero, however the angular rate and guidance quaternion are still given to the controller. The results of this test clearly show that the average guidance error is heavily influenced by the implemented feed forward control and that the controller based on simplified attitude dynamics is only coarsely capable of following a guidance trajectory.

6.3 Demonstration scenarios

This section demonstrates the overall capabilities of the Facility for Attitude Control Experiments (FACE). It is not the intention to show the limits of the laboratory's possibilities, but the common uses for which it was built. Two designed example missions will show the realistic simulation of a satellite maneuvering according to a defined profile, demonstrating a reliable usage for simulation with longer durations with repeatable measurement results. The implementation of the basic attitude control system (ACS) developed in the previous sections is adequate to verify the functionality, showing the potential of the FACE when employing more advanced algorithms and additional hardware.

6.3.1 Successively approaching a series of attitudes

The first mission was a series of attitudes that were approached successively with the feed forward guidance and a linear controller. All target quaternions had a rotation angle of 15° with varying rotation axes. At the beginning all three body fixed axes were rotated in different combinations, followed by rotations around only two of the axes, and finally around each seperate axis alone. The first diagram of Fig. 6.6 shows the profile of the mission controlled by the PD controller; the second diagram shows the error angles with respect to the guidance. The guidance is followed with a very rough profile, where the error oscillates around the trajectory.

The same mission profile was used a second time, but this time the maximal acceleration of the feed forward guidance from section 6.2.4 is decreased from $0.043^\circ/\mathrm{s}^2$ to $0.007^\circ/\mathrm{s}^2$, reducing the step in control torque. This led to a longer duration of the mission and a

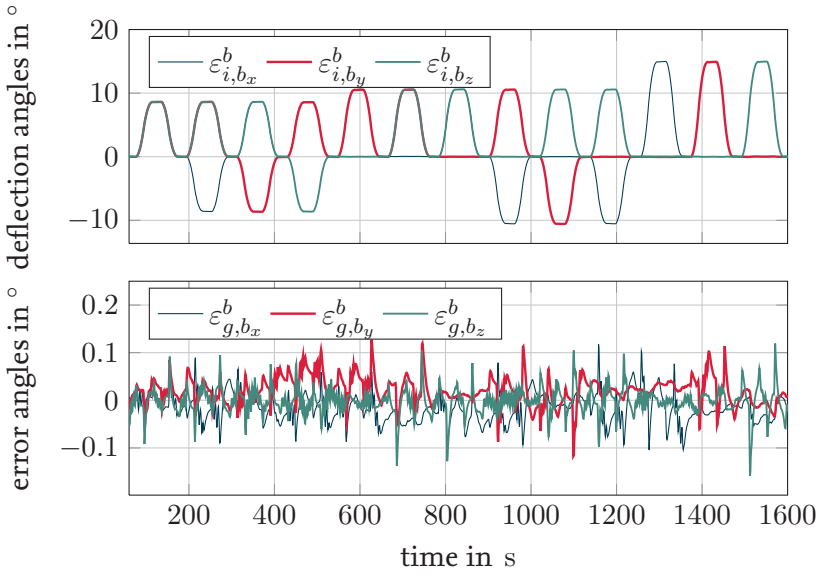


Figure 6.6: Guiding through a series of target attitudes ($\dot{\omega}^b_{i,b_{\max}} = 0.043°/s^2$)

different shape of the deviation from the guidance in the second diagram of Fig. 6.7. This time the controller did not overreact as much as in the last test, but an oscillation around the zero line remained, even if with a smaller amplitude, in particular, the oscillation for the z axis (ε^b_{g,b_z}) showed a high frequency. These oscillations could be reduced to some extent by changing the value for the punishment of the control signal $\underline{R}_w$ in the LQR optimization. The longer mission was also tested with the PID controller, resulting in a significantly better accuracy compared to the PD controlled experiment. The peak at 2700 s is owed to a short overload of the control computer (Windows™/MATLAB™Simulink™) during which the OBC did not control the system. It may be the case that the difference between both controllers lies in the fact that the PD controller is not able to compensate for a constant disturbance torque resulting from the misaligned c. m.. This could explain the drift in the error angles of the PD controller if the c. m. is influenced by temperature changes during the long experiment, which the PID controller compensated for with the additional control deviation $\underline{e}$.

This first mission demonstrated the reliable functioning of the experimental satellite during simulation times of up to one hour. The shown repeatability of the maneuvers is very important for comparing different ACSs to each other, or for testing different approaches to the design of controllers.

6.3.2 Ground station guidance

Performing experiments in the FACE requires considerations on how to avoid the limitations of the air bearing table while still simulating realistic satellite missions. Tracking a ground station from an orbit is a maneuver with large slew angles over a simulation time

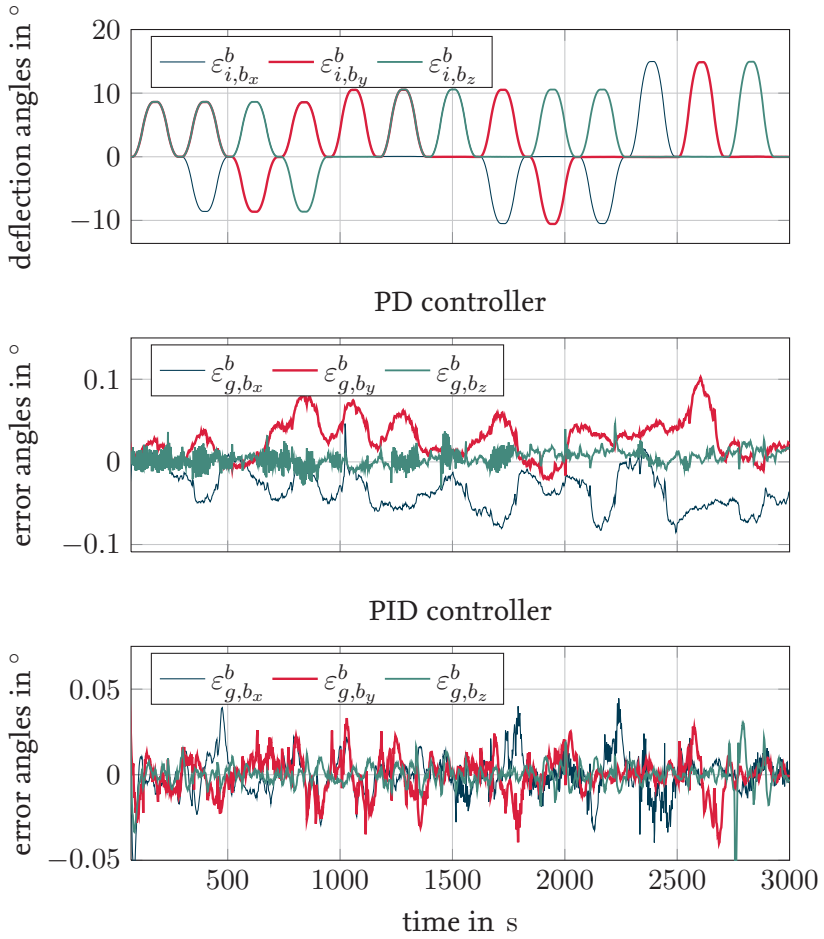


Figure 6.7: Guiding through a series of target attitudes ($\dot{\omega}^b_{i,b_{\max}} = 0.007°/s^2$)

of a few minutes. It would be possible to simulate only a small part of the whole maneuver, or to find a reference system that allows the experimental satellite to execute most of its rotation around the vertical axis, which underlies no restrictions in its degree of freedom.

For the demonstration of a ground station guidance mission, the latter approach was chosen, using the PID controller to improve the guidance accuracy. This approach is limited to an orbit where the satellite passes the ground station directly above or at least in a small window around the zenith. Aligning a satellite-mounted instrument with a target defines no complete attitude. If the instrument points to the ground station, the satellite is still free to rotate around the alignment axis. A second constraint is necessary to define a unique attitude for the ACS to follow. The additional degree of freedom is typically used to align the solar panels towards the Sun to generate as much electrical energy as possible.

Orbit and frames

This experiment uses an orbit propagation to simulate the orbit of the satellite; at the same time, the air bearing table is used for the rotational degrees of freedom. The main reference frame is the ECI frame i. A second reference frame r is necessary, which is located at the propagated position of the satellite's c. m. with the same alignment as the inertial frame (ECI). If this frame r were used as the simulation frame in the laboratory, the maneuver would not be restricted to the vertical axis of the satellite, hence a third frame is defined that has a constant rotation towards the reference frame r. This frame o uses a coordinate system in which the x and y axes lie in the orbit plane and the z axis is perpendicular to the plane. This frame is used inside the laboratory because a direct pass above the ground station will result in a rotation around the z axis whereas all other rotation angles are small (Imagine the air bearing stator standing below the orbit plane from Fig. 2.3, a depiction of this correlation can be found in Fig. A.1). The quaternion $\mathbf{q}_o^b$ describes the orientation of the body fixed frame b with respect to the defined orbit frame o, or the laboratory frame, respectively.

The orbit propagation is done on the basis of the orbital parameters from section 2.1.1. The orbit is defined by a close pass of the ground station and a Sun vector that is almost perpendicular to the resulting orbit plane.

$$\begin{array}{lll} a = 7178\,\mathrm{km} & e = 0 & i = 99.5^\circ \\ \omega = 0^\circ & \Omega = -81^\circ & t = 06.10.2009\ 05{:}00\ (+800\,\mathrm{s}\ \text{for direct pass}) \end{array} \tag{6.45}$$

These orbit parameters are used to simulate two orbits that pass the ground station at different distances. Fig. 6.8 shows the two footprints of the passes in the vicinity of the ground station, one directly passing over the ground station located in Bremen, and one

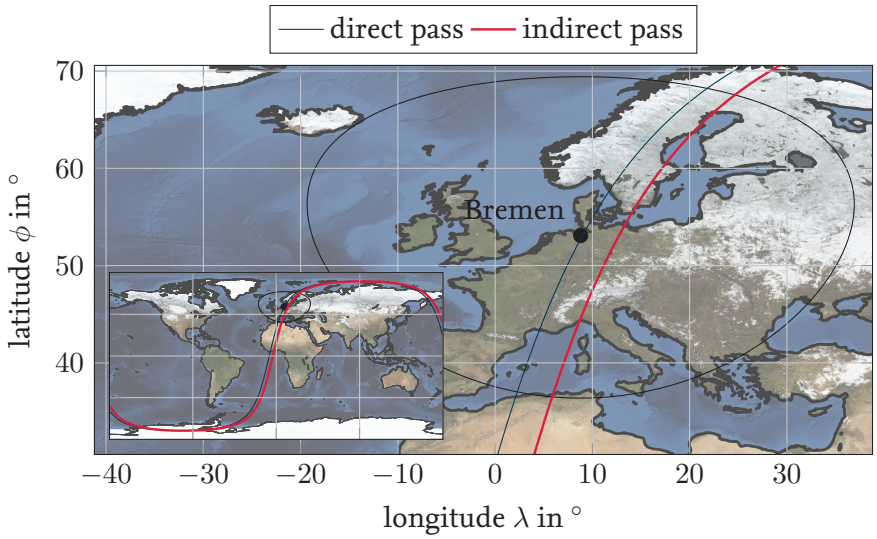


Figure 6.8: Ground track of the simulated satellite in the vicinity of a ground station

with an offset of a few hundred kilometers. In addition, the picture shows the area in which the satellite's *line-of-sight*, viewed from the ground station, has an elevation of more than $15°$ above the horizon.

Orientation towards the ground station

The next step is to determine the desired attitude for the satellite to align its instrument with the ground station and to align the solar panels with the Sun as well as possible. The *algebraic method* from [2] is used to calculate a transformation matrix of the reference frame r with respect to the ground station gs. Two direction vectors are required for this: the direction from the reference frame towards the ground station $\underline{r}^r_{r,gs}$, and the direction towards the Sun $\underline{r}^r_{r,sun}$. These vectors are used to calculate an orthonormal basis, and thus the transformation matrix $\underline{\underline{T}}^r_{gs}$ and the quaternion $\mathbf{q}^r_{gs}$, using the following algorithm:

$$\underline{q} = \frac{\underline{r}^r_{r,gs}}{r^r_{r,gs}} \,,\; \underline{r} = \underline{q} \times \frac{\underline{r}^r_{r,sun}}{r^r_{r,sun}} \,,\; \underline{s} = \underline{q} \times \underline{r} \tag{6.46}$$

$$\underline{\underline{T}}^r_{gs} = \begin{bmatrix} \underline{q} & \underline{r} & \underline{s} \end{bmatrix} \tag{6.47}$$

The direction vectors are determined with the help of available implementations for the position and attitudes of celestial bodies. The ground station is located in Bremen ($\lambda \approx 53.1°\mathrm{N}$, $\beta \approx 8.8°\mathrm{E}$). In cartesian coordinates of the ECEF frame e, this corresponds to

$$\underline{r}^e_{e,gs} = \begin{bmatrix} 3794.1 & 587.4 & 5076.1 \end{bmatrix}^{\mathrm{T}} \mathrm{km}\,. \tag{6.48}$$

The transformation ECEF $\rightarrow$ ECI is a complex function of the time, thus the vector $\underline{r}^i_{i,gs}$ varies over time. The sought alignment vector $\underline{r}^r_{r,gs}$ is the difference of the satellit's propagated position and the ground station's position:

$$\underline{r}^r_{r,gs} = \underline{r}^i_{i,r} - \underline{r}^i_{i,gs} \tag{6.49}$$

The implementation for the Sun's location is also based on the time and returns the value $\underline{r}^i_{i,sun}$, which, due to the distance between Sun and Earth, is approximately $\underline{r}^r_{r,sun}$. This concludes the attitude quaternion $\mathbf{q}^r_{gs}$ of the satellite's location with respect to the ground station.

The satellite must be aligned towards the ground station according to the instrument that has to target it. In this case, a laser pointer on the lower side of the assembly station was chosen to represent this instrument. The algebraic method is used again to construct a basis for the correct alignment. The main direction is defined by the instrument's line-of-sight $\underline{r}^b_{b,inst} = \begin{bmatrix} -1 & 1 & 0 \end{bmatrix}^T$ with respect to the body frame, with a second constraint of

the solar panel's surface normal $\underline{r}^b_{b,panel} = \begin{bmatrix}0 & 0 & 1\end{bmatrix}^T$.

$$\underline{q} = \frac{\underline{r}^b_{b,inst}}{r^b_{b,inst}}\ ,\ \underline{r} = \underline{q} \times \frac{\underline{r}^b_{b,panel}}{r^b_{b,panel}}\ ,\ \underline{s} = \underline{q} \times \underline{r} \tag{6.50}$$

$$\underline{\underline{T}}^b_{inst} = \begin{bmatrix}\underline{q} & \underline{r} & \underline{s}\end{bmatrix} = \begin{bmatrix} -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\ 0 & 0 & -1 \end{bmatrix} \tag{6.51}$$

Because the instrument is fixed on the assembly station, the resulting quaternion $\mathbf{q}^b_{inst}$ stays constant during the simulation.

Orbit frame

The orbit frame o is defined during the first step of the orbit propagation. As above, the transformation between the frame r and o results from two known vectors. These vectors are the position and the velocity of the satellites frame r in inertial (ECI) coordinates:

$$\underline{q} = -\frac{\underline{r}^i_{i,r}}{r^i_{i,r}}\ ,\ \underline{r} = \underline{q} \times \left(\frac{\underline{v}^i_{i,r}}{v^i_{i,r}} \times \frac{\underline{r}^i_{i,r}}{r^i_{i,r}}\right)\ ,\ \underline{s} = \underline{q} \times \underline{r} \tag{6.52}$$

$$\underline{\underline{T}}^r_o = \begin{bmatrix}\underline{q} & \underline{r} & \underline{s}\end{bmatrix} \rightarrow \mathbf{q}^r_o \tag{6.53}$$

Guidance

The ACS aligns the varying guidance quaternion $\mathbf{q}^{gsg}_o$ with the experimental satellite's attitude $\mathbf{q}^b_o$, and follows the ground station over the course of the experiment. The guidance quaternion with respect to the reference frame r is calculated as follows:

$$\mathbf{q}^{gsg}_r := \mathbf{q}^b_r = \mathbf{q}^{gs}_r \mathbf{q}^b_{gs} = \mathbf{q}^{gs}_r \left(\mathbf{q}^{inst}_{gs} \mathbf{q}^b_{inst}\right) = \mathbf{q}^{gs}_r \mathbf{q}^b_{inst} \tag{6.54}$$

The quaternion $\mathbf{q}^b_{gs}$ between the ground station and the body fixed frame is calculated using the known quaternion $\mathbf{q}^b_{inst}$ and the quaternion $\mathbf{q}^{inst}_{gs}$, where the latter is the identity quaternion because the instrument should point directly at the ground station. With the known rotation between reference and orbit frame, the guidance quaternion is

$$\mathbf{q}^{gsg}_o = \mathbf{q}^r_o \mathbf{q}^{gsg}_r\ . \tag{6.55}$$

All calculations and simulations from above suffice to follow an imaginary ground station in the FACE. At the beginning of the experiment the experimental satellite is in its home position, which may differ greatly from the attitude towards the ground station. For this reason, the slope limiter from section 6.2.4 is used to guide a transit slew maneuver to smoothly rotate the satellite towards the ground station attitude $\mathbf{q}^{gsg}_o$. At the start of this slew maneuver, the target quaternion $\mathbf{q}^t_o$ is set to the current ground station attitude.

During the guidance maneuver, the attitude towards the ground station slowly changes, and a difference quaternion is calculated to include this drift in the guidance:

$$\mathbf{q}_t^{gsg} = \mathbf{q}_t^o \mathbf{q}_o^{gsg} \tag{6.56}$$

The guidance quaternion $\mathbf{q}_o^g$ from the slope limiter is multiplied by this difference quaternion, resulting in the new ground station guidance, which is the superposition of the slope limiter guidance and the calculated trajectory to track the ground station:

$$\mathbf{q}_o^{gsg'} = \mathbf{q}_o^g \mathbf{q}_t^{gsg} \tag{6.57}$$

At the end of the transit maneuver, the quidance $\mathbf{q}_o^g$ is equal to the target $\mathbf{q}_o^t$, and $\mathbf{q}_o^{gsg'} = \mathbf{q}_o^{gsg}$.

For a better guidance accuracy, the angular rate and acceleration are required for the feed forward control. The guidance $\mathbf{q}_o^{gsg}$, rearranged in a 3×4 matrix, and the quaternion's derivative $\dot{\mathbf{q}}_o^{gsg}$ are necessary to compute the angular rates $\underline{\omega}_{o,gsg}^b$ according to [13]:

$$\underline{\omega}_{o,gsg}^b = \begin{Bmatrix} q_4 & q_3 & -q_2 & -q_1 \\ -q_3 & q_4 & q_1 & -q_2 \\ q_2 & -q_1 & q_4 & -q_3 \end{Bmatrix}_o^{gsg} \left\{ \dot{\mathbf{q}}_o^{gsg} \right\} \tag{6.58}$$

The superposition gsg' of both guidances as an open-loop control signal is

$$\mathbf{q}_o^{gsg'} \tag{6.59}$$

$$\underline{\omega}_{o,gsg'}^b = \underline{\omega}_{o,g}^b + \underline{\omega}_{o,gsg}^b \tag{6.60}$$

$$\dot{\underline{h}}_{b,\mathbf{W}_{gsg'}}^b = \dot{\underline{h}}_{b,\mathbf{W}_g}^b + \dot{\underline{h}}_{b,\mathbf{W}_{gsg}}^b = \dot{\underline{h}}_{b,\mathbf{W}_g}^b - \underline{\underline{I}}_{\mathbf{B}}^{b\,-1} \frac{\mathrm{d}_{gsg} \underline{\dot{\omega}}_{o,gsg}^b}{\mathrm{d}t} \approx \dot{\underline{h}}_{b,\mathbf{W}_g}^b \,. \tag{6.61}$$

The negative ground station guidance torque $\dot{\underline{h}}_{b,\mathbf{W}_{gsg}}^b$ can be calculated using the satellite's known moment of inertia and the numerical derivative of the angular rate. However, since the slow maneuver only requires very small accelerations which the controllers are capable of compensating for, it can be neglected. The ACS is now able to guide the experimental satellite towards the ground station from an arbitrary starting attitude, and afterwards to follow the ground station during the pass, with a very smooth transit between both maneuvers due to the superposition of both guidances.

Simulation and experiment

The test mission started with a slew maneuver from the home position of the air bearing table to an intermediate position. When the satellite reached an elevation of $15°$ above the horizon, relative to the ground station, the superimposed transit maneuver began. The maneuver led over to the pure ground station guidance, and when the elevation fell below $15°$ after the pass, the experimental satellite moved back to its home position with an intermediate position in between.

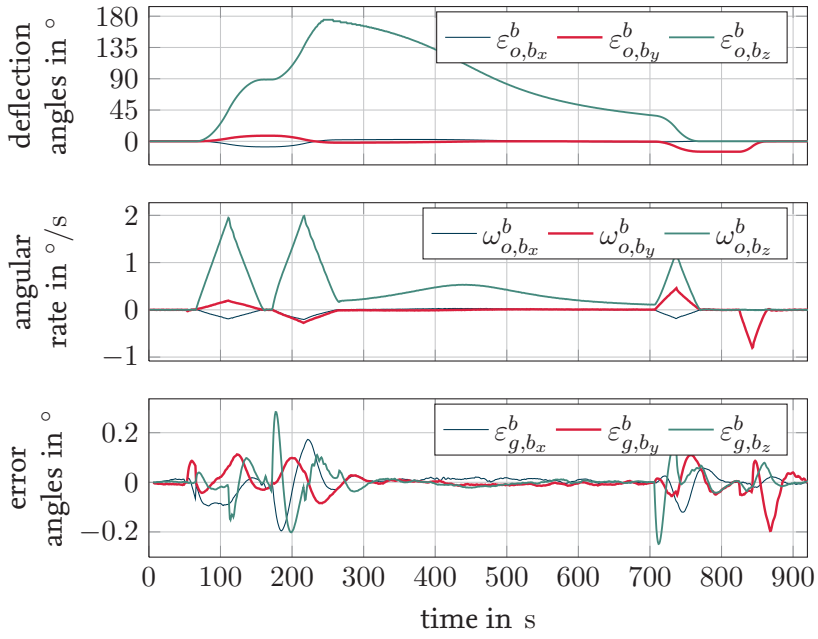


Figure 6.9: Ground station guidance from an orbit that directly passes above the ground station

Fig. 6.9 shows the measurements during the experiment for the direct pass of the ground station. The first 300 s were the maneuvers to align the laser pointer with the ground station, which was then tracket up to approximately 700 s, after which the slew maneuver towards the home position started. The first diagram shows the deflection angles calculated from the satellite's attitude, which is why the curve for the z deflection creases at 180° even though the maneuver included a full rotation around the z axis. The angular rates for the mission are depicted in the center diagram, with the error angles on the bottom. The errors during the ground station guidance are more important than the errors during the transit maneuvers. After the oscillations were eliminated by the PID controller, a very smooth and accurate guidance was accomplished.

The results for the indirect pass from Fig. 6.8 are very similar to the direct pass, except for the deflection angles and angular rates of the two other body fixed axes, which are much more pronounced in Fig. 6.10 compared to the direct pass.

Fig. 6.11 is a magnification of the deviation from the guidance trajectory during the important interval. The noisy signals in this magnification are possibly related to the drift of the integrated gyro values that have a very similar appearance if the assembly station is held in position by the pneumatic actuator. The accuracy of both passes shows that a precise implementation of an ACS for guiding purposes was developed. This demonstrates the capabilities of the FACE in its current state, which can be improved by additional hardware and better algorithms. In particular, an external reference system is necessary

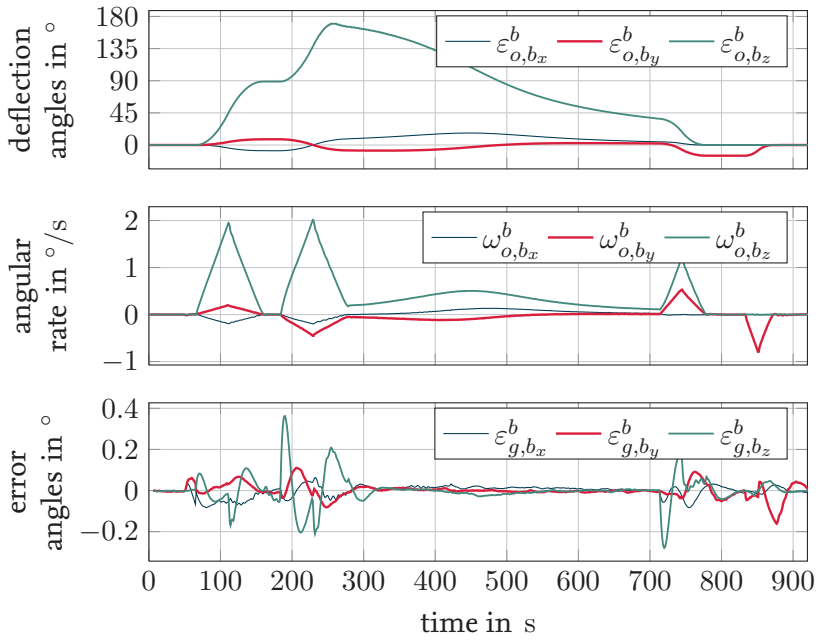


Figure 6.10: Ground station guidance from an orbit that does not directly pass the ground station

to determine the absolute accuracy, not only the accuracy with regards to the inertially measured attitude.

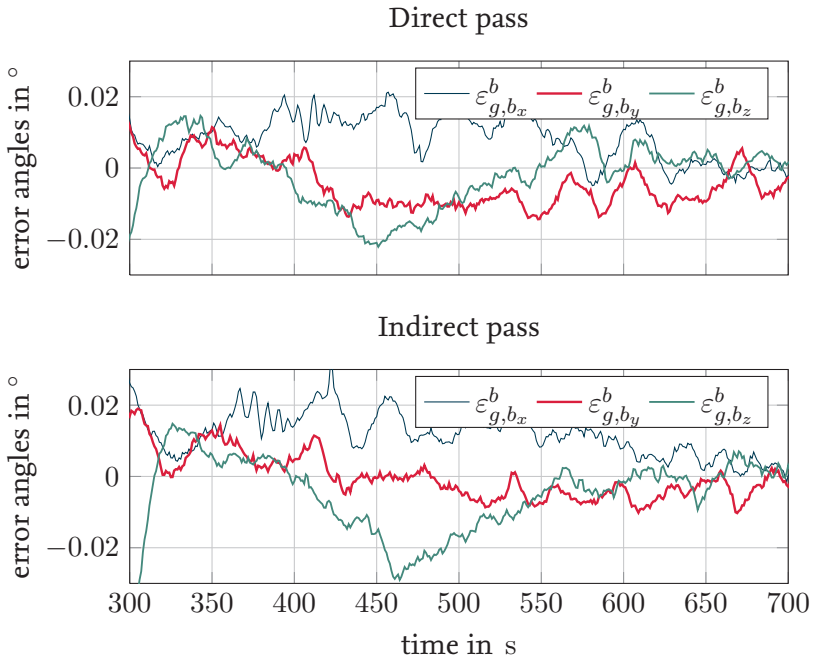


Figure 6.11: Accuracy of the ground station guidance mission

7 Implementation procedure

This chapter is a guideline for the implementation and integration of new hardware on the experimental satellite in the Facility for Attitude Control Experiments (FACE). It is also intended to be a basic manual for bringing a new or modified attitude control system (ACS) into service. The general procedure is given in the flow chart in Fig. 7.1.

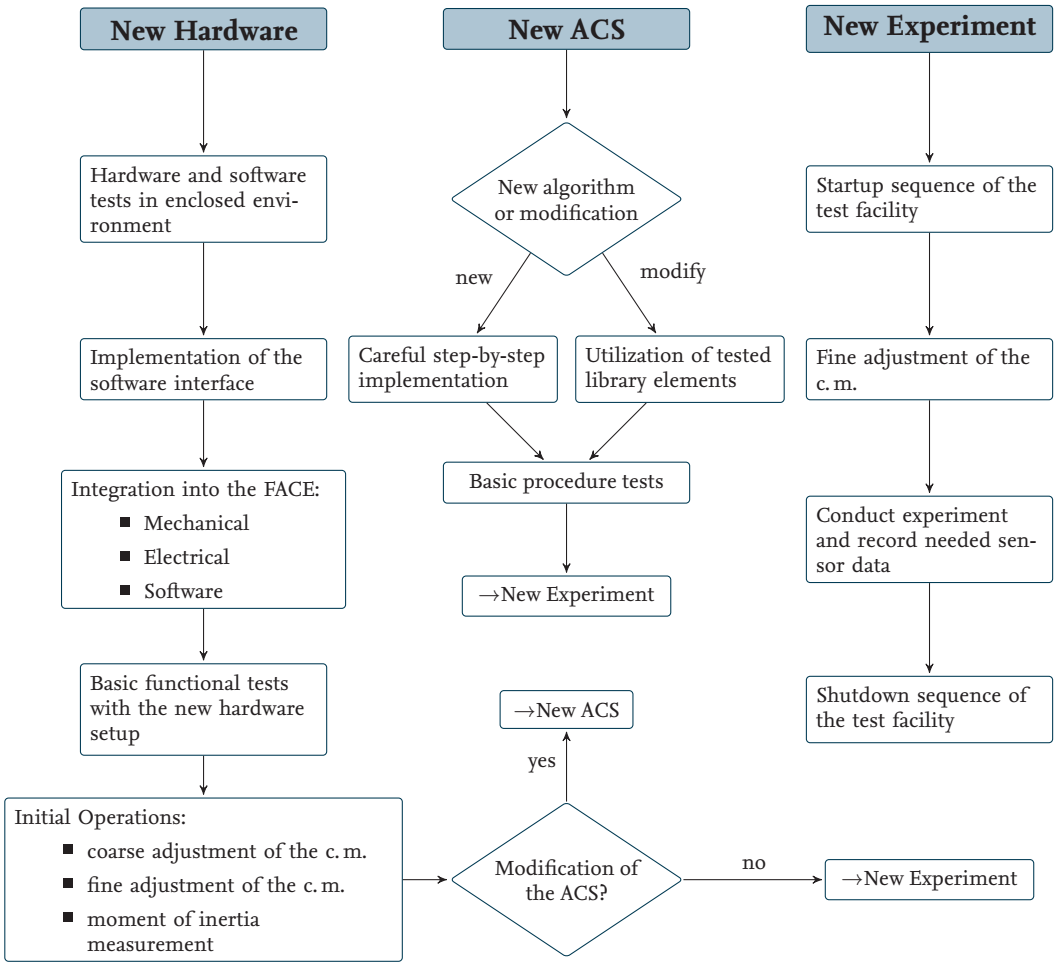


Figure 7.1: Implementation procedure

Problems that occurred during the setup and the initial operation of the laboratory will be addressed, and hints given for performing experiments safely and efficiently. Further information about the test facility can be found in the official documentation [13, 14, 4].

7.1 Prerequisites

The current state of the laboratory is given as an overview in Fig. 7.2. It separates the

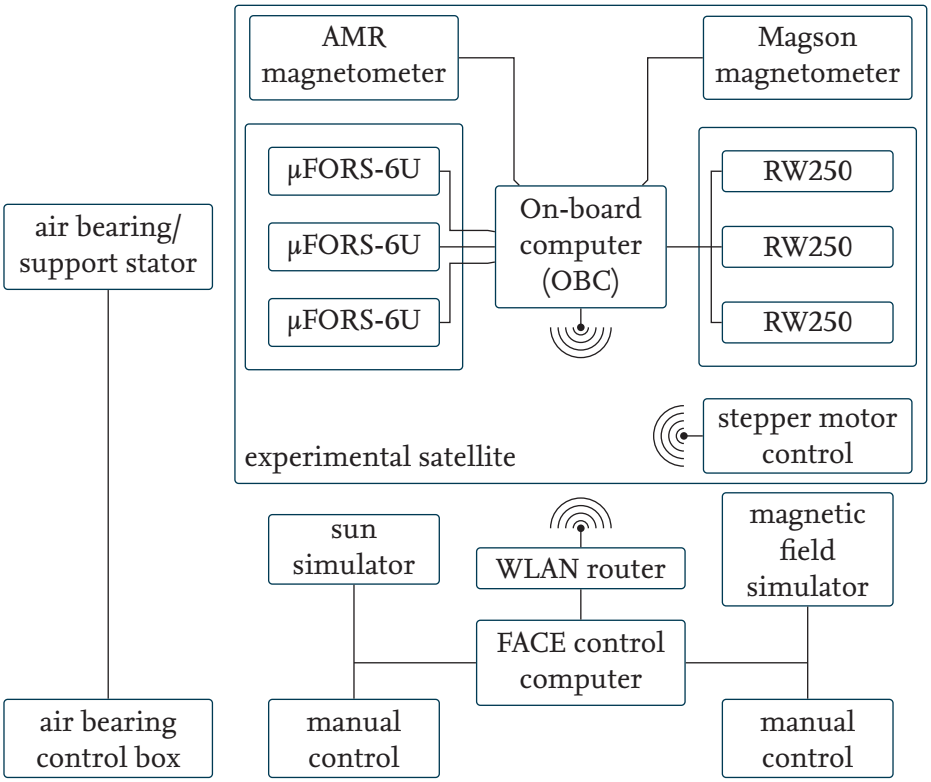


Figure 7.2: Overview of the FACE with its main components

experimental satellite's hardware from the control hardware and the environment simulation. The central component of the laboratory is the control computer, whereas the central component of the experiential satellite is the on-board computer (OBC). Both are connected to a WLAN for monitoring the course of an experiment, or for copying new programs to the OBC and simulation data back to the control computer. The hardware for the air bearing and its safe usage is decoupled from the rest of the FACE. The *air bearing control box* is positioned at the workstation to quickly abort an experiment from a safe distance from the air bearing and the possibly moving assembly station.

7.1.1 Hardware

The hardware for use in an ACS is directly connected to the OBC, which is the only device that can communicate with the actuators and sensors. The satellite possesses three WLAN devices of which two are currently in use.

OBC An internal interface can connect to the laboratory's WLAN; it will receive the IP address 192.168.124.42 from the WLAN router (192.168.124.1).

EIA-232 to WLAN converter This converter tunnels the EIA-232 communication of the control board for the stepper through a WLAN. It has a fixed IP address (192.168.124.35) that can be changed with a provided web interface (192.168.124.35, port 80). The tunneled serial data is accessible via port 4660.

LAN to WLAN converter This converter is not currently in use; it was provided for the case that the OBC did not hold WLAN capabilities.

The encryption for the wireless communication is the very unsafe standard *WEP*, which is a security threat that needs to be addressed by using a safer encryption standard, at least for the communication between OBC and control computer. Refer to [1] for the passwords and user names of the devices.

The WLAN converter and the steppers and their control hardware are usually turned off to reduce the power consumption and prevent magnetic disturbances from the stepper motors. A remote control is used to switch these devices on and off.

The environment simulation consists of the sun simulator and the magnetic field simulator; both could be tuned manually without the use of the control computer. The sun simulator can be controlled via a DMX interface, for which only a preliminary software interface is available at the moment. The power supplies for the magnetic coils are connected to each other and are controlled by the computer through a *GPIB* interface [15].

7.1.2 Software toolchain

Several software applications are required for working in the laboratory. The software for an ACS is developed and built on the control computer, or any other computer with appropriate equipment. The executable program is copied to the OBC, where it runs autonomously or monitored by the control computer.

In the best-case scenario, the developer uses the MATLAB™/Simulink™ environment and all other steps to create the executable happen automatically. For all other cases it is good to know how the process works, and how and where to intervene. The user develops their new algorithm in Simulink™ with all advantages of a high level development approach, without having to take care of basic variable assignment, memory allocation, and so forth.

After a new algorithm is implemented, the user starts the build process by hitting the shortcut key combination *CTRL + 'B'*. This starts the add-on Real-Time Workshop™ (RTW) (recently renamed to Coder™) that generates platform independent C source code of the Simulink™ model. The source code is merged with a template program chosen for the target operating system and architecture, in this case for QNX and a standard $32\,\mathrm{bit}$ (x86) computer.

The resulting software tree is then processed by the *qcc* compiler (a proprietary branch of the gnu compiler collection (GCC) modified for the QNX RTOS) that produces an executable application (only executable on the target platform). If the build process is successful, a second user interaction starts the script *HWI_upload* that uploads the program to the OBC, starts it, and connects the program to Simulink™ in order to monitor the

newly built application. At this point it can be started with the play button in the menu bar. This concludes the short summery of the software integration process with Simulink™.

Two separate applications were delivered by the main manufacturer of the test facility.

ACS Trimmung A small application for a basic control of the fine adjustment mechanisms over WLAN.

ACS Magnetfeld This program connects to the power supplies over a *GPIB* interface to control the magnetic field of the magnetic simulator by changing the values manually, or automatically with a table of a calculated time series of the magnetic field strength components.

A hardware programming device is available for the LITEF μFORS-6U rate gyros that is controlled by the software *IMK_3*. This device is connected to the computer and is able to reprogram a gyro to output the sensor values in different ranges, different units, or with different sample times.

Other useful software is:

ssh/telnet are terminal clients to remotely work on the OBC; the OBC has neither a keyboard nor a display when mounted onto the assembly station, making a remote shell the only way to access the operating system.

scp is a command line program for copying files between computers. For example, measurement data that was recorded locally on the OBC during an experiment is accessible through this application (*Filezilla*, if a graphical user interface is necessary).

QNX Momentics is the development environment from the QNX manufacturer, providing a highly sophisticated debugging console for remotely monitored embedded systems.

7.2 New hardware

The integration of new hardware for use on the experimental satellite requires several considerations, and at all times caution. When handling hardware, electrostatic discharge (ESD) precautions must be taken to protect the device from damage, in particular when working on the wiring. This section is split into multiple parts to address all important subjects separately.

7.2.1 Software

The software development of an interface driver for a new hardware should start in a laboratory environment and not on the experimental satellite. A power supply with the ability to monitor the power consumption is good for having an indication whether the device is working properly, possibly concluding the operation mode the device is currently works in, for example, power modes like *standby*. A fast fuse for the case of a malfunction is another advantage of some laboratory power supplies.

Within this environment the software test should start with a very basic check of the communication: sending a command that should be recognized by the device and waiting for the corresponding acknowledgment. If the communication link works, the actual communication protocol can be implemented, which is usually defined in the user manual of the device. During this implementation it might have an advantage to start coding a subset of the protocol's instructions, and excessively test the communication, possibly testing expected error cases. Afterwards, the remainder of the protocol can be programmed for areas of lower importance, such as changing the power states, or checking the housekeeping data.

If tested implementations are already available for parts of the developed software they should be used to decrease the amount of work and increase the reliability through the use of proven code. If the implementation is made for the Simulink™ environment, the standard communication channels, such as serial ports (EIA-232, EIA-422, etc.) and TCP/IP sockets, are already available and can be utilized instantly.

7.2.2 Mechanical

The integration of a new hardware on the satellite starts with a temporary fixation of the device. Even initial tests should be made with a securely fastened device, because a small slip or a shaking mass can have a considerable impact on the stability of the experimental satellite. A well chosen position of a new device is mandatory. It has to be considered whether the position can be chosen arbitrarily, such as for the rate gyros, or whether a special location on the assembly station is necessary for the functioning of the device, for example, magnetometers should be positioned in the vicinity of the homogeneous magnetic field of the Helmholtz setup. If possible, the hardware should be mounted in such a way that the overall c. m. stays very low, because additional counter weights may exceed the maximal load of the air bearing, or increase the moment of inertia, which may contradict the requirements of the test setup. It may be unavoidable to move other devices on top of the assembly station in order to make room for the new hardware. This should be done very carefully, because it increases the effort to readjust the air bearing table.

After new hardware is integrated on the experimental satellite, the setup must be put into operation by following the initial procedure in section 7.3 prior to performing new experiments.

7.2.3 Electrical

The electrical wires for communication and power supply need to be tightened very well, preventing them from slipping during an experiment. When changing the wiring of the satellite, it should be observed that even cables noticeably influence the c. m.

The experimental satellite has its own battery operated on-board power grid for several voltage levels. The different levels are provided by output sockets, which only allow a specific maximal power consumption; exceeding this limit could damage the power box [13]. Most of the sockets are already in use; connecting a new device to an already used socket

is not a problem as long as it is ensured that the maximal combined power consumption of all connected hardware does not exceed the given limits.

7.3 Initial operation

7.3.1 Adjusting the center of mass

At the very least, the c. m. has to be newly adjusted every time the setup of the assembly station is changed, in order to guarantee that the experimental satellite is free of disturbance torques. The coarse adjustment is done by moving the brass weights until a stable configuration is found. The first tests showed that successively attaching additional weights is not a good approach for adjusting a newly installed setup. A better way is to symmetrically arrange weights on the beams with margins for moving them upwards and downwards. If a stable configuration of the table is found, the weights can be displaced in small millimeter steps to move the c. m. below the pivot point until the fine adjustment mechanisms for the z axis is enought to lift the c. m. to the pivot point. The small movable weights inside the x and y mechanisms should be located somewhere in the middle of the corresponding axis, for which the *ACS Trimmung* application could be used.

The rest of the alignment process is automatically conducted by the Simulink™ implementation *FACE_HWI_autoadjustment.mdl*, which controls the motion of the fine adjustment weights along the body fixed axes (5.1). Executing the file *everything.m* configures the necessary environment. In this implementation it is necessary to choose a basic profile for the *sequence* subsystem inside the *automated guidance* block:

auto_adjust_center_of_mass The standard profile to adjust the c. m. in four steps. The first and fourth step perform the alignment in the x-y plane, the second and third handle the remaining z direction.

auto_adjust_center_of_mass_xy_only The adjustment is only executed for the x-y plane.

auto_adjust_center_of_mass_start_steppers_immediately The complete adjustment is executed for all three spatial directions without waiting for a steady state after a transit maneuver. This should be used the first time after a major change in the setup was made, because otherwise the reaction wheels can saturate rapidly, and by choosing this option the steppers are activated immediately without wasting time.

Prior to the start of the application, the fine adjustment and its control hardware have to be switched on with the remote control. The pneumatic actuators can be retracted after the first $50\,\mathrm{s}$ of simulation time, during which the calibration of the rate gyros happens. The simulation finishes with the stepper velocity being set to zero after the last transit maneuver from the chosen profile. The velocity can be monitored in the *fine_adjustment_control* subsystem. The simulation can be aborted at any time by toggling the *enable_guidance* switch that lets the assembly station rotate back to its home position. In this case it is necessary to switch off the fine adjustment hardware with the remote control.

7.3.2 Measuring the moment of inertia

The estimation of the satellite's moment of inertia consists of two parts: measurement data must be recorded during a simulated mission, and the recorded data must be filtered and evaluated. The data is recorded during an automated sequence that is tuned for the maximal possible angular acceleration around all body fixed axes, and for exhausted total deflection angles. The Simulink™ model for this is *FACE_HWI_inertiameasurement.mdl*. After the simulation time passes the $50\,\mathrm{s}$ mark, the pneumatic actuators have to be retracted, releasing the experimental satellite. At any time, the mission can be aborted by switching off the guidance.

The text file *everything.m* is utilized for tuning the process, and additionally for loading all necessary libraries. Three variables in this file influence the guidance sequence in major ways:

```
% assumed moment of inertia
MoI=eye(3,3)*40;
guidance.maxAngularVelocity=10*pi/180;
guidance.maxAngularAcceleration=0.0125;
```

The two guidance values, the maximal allowed total angular velocity and acceleration, limit the *slope limiter* (section 6.2.4) for the trajectory. If the setup of the experimental satellite was changed, the value for the acceleration should be decreased, and can then be carefully increased over multiple runs of the automatic sequence. If the assumed moment of inertia (in this example $40\,\mathrm{kgm}^2$ for all axes) and the maximal acceleration are not chosen well, the experimental satellite cannot be correctly guided by the utilized PD controller. This could result in exceeding the target attitudes that are very close to the mechanical end stops. To prevent damage, small values should be used for the guidance variables, and choosing an estimated moment of inertia matrix should err on the side of choosing values that are too high. The values can be tuned during several runs, resulting in high measured accelerations caused by high control torques from the reaction wheels.

The recorded data is saved on the OBC in a file called *simulationdata.mat*; it is sampled with at a frequency of $200\,\mathrm{Hz}$ and its structure is described in Tab. 7.1. The automatic estimation of the moment of inertia is done by the script *estimate_MoI.m* that includes the following important lines:

```
% data file
load('simulationdata.mat')
% Initial MoI
% [I_11 I_22 I_33 I_12 I_13 I_33]
I=[30 30 30 3 3 3]';
% Optimize over time range
myrange=[400:1250];
% Delay range
delayrange=[-200:50:200];
```

The location of the newly recorded data has to be customized here, and the initial guess for the six independent elements of the moment of inertia matrix is necessary for the

Table 7.1: Structure of the recorded data

rows	elements	variable
1	1	time
2-5	4	attitude quaternion
6-8	3	filtered angular rates
9-11	3	raw angular rates
12-14	3	error angles towards target attitude
15-17	3	error angles towards guidance attitude
18-20	3	derivative of the reaction wheels' angular momenta
21-23	3	reaction wheels' angular momenta
24-29	6	control deviation
30-32	3	control torque
33(-35)	1 or 3	stepper velocities or zero signal if not used

optimization (section 5.2). After the data is filtered and resampled to $5\,\mathrm{Hz}$ it is necessary to specify an interval (*myrange*) over which the optimization should be conducted. In the example given above, the interval starts at $80\,\mathrm{s}$ and ends at $250\,\mathrm{s}$. The last important parameter is a range of delays to find the best shift between the two sensors that best satisfy the modeled dynamics (Fig. 5.9). These values are given for the unfiltered $200\,\mathrm{Hz}$ data set; A negative values moves the reaction wheels' data back in time. The computation time for this script is significant, because an extra optimization is executed for each element of the delay range. The results are stored in two arrays:

```
I_array
cost_array
```

The lowest cost identifies the moment of inertia that best matches the measurement data.

7.4 Building an attitude control system

The Simulink™ environment has to be configured for the QNX real-time target when a new ACS is to be implemented. The basic configuration for a blank Simulink™ model to work on the OBC starts in the options dialog via: *Tools* → *Real-Time Workshop* → *Options*. In the newly opened window it is necessary to change to the correct target template: *Real-Time Workshop* → *Target selection* → *System target file* → *Browse*. This opens a list of available targets from which the *qnx.tcl* element should be chosen. If the list does not include the correct target, the QNX real-time target template files have to be copied to:

```
%PathToMATLAB%\MATLAB\%Version%\rtw\c\qnx
```

For monitoring an experiment in real-time, the so-called *external mode* has to be configured. Two options in *Real-Time Workshop* → *Interface* → *Data exchange* → *External mode* → *Host/Target interface* have to be adjusted. First, the *Transport layer* is *tcpip*, and second, a

variable name is put into the *Mex-file arguments* field: *HWI.TargetIP* (or the IP address of the OBC, if the HWI scripts from the next section are not used).

The simulation time should start at *0.0* (*Solver* → *Simulation time* → *Start time*), and end at a chosen time, or *inf* if the user has to stop the simulation manually (*Solver* → *Simulation time* → *Stop time*).

The solver for the simulation is set to a fixed step algorithem (*Solver* → *Solver options* → *Type* → *Fixed-step*), and a sample time must be defined, which is used as the maximal possible frequency on the real-time target for executing a task: *Solver* → *Solver options* → *Fixed-step size*. After applying the new configuration and closing the options window, the value *External* has to be set in the *simulation mode* pull-down menu in the menu bar of the Simulink™ model.

The build process of a Simulink™ model is started via *Tools* → *Real-Time Workshop* → *Build Model*. The easiest way to copy the resulting executable to the OBC and to start the application is to use the predefined scripts that are described in the next section 7.5.

The executed application on the OBC can be controlled and monitored in real-time during an experiment. It is possible to use *switches* and change *constants*, but during the time a constant is changed in a pop-up window, for some reason, the simulation (including the on-board real-time application) stalls. Aside from that, the simulation is also influenced if Windows™ is very busy.

It is advised to put a new ACS, or major changes of a model, into operation on a step-by-step basis, because the debugging process is very time-consuming due to recompiling, restarting, and recalibrating.

7.5 Libraries

Two Simulink™ libraries are available for supporting the user during their development of an ACS. One library is a set of software interfaces for the simple control of external attitude control hardware (chapter 4) that is currently available for the facility. The other library is a set of predefined algorithms that are commonly used in most implementations developed to this point.

7.5.1 HWI_lib

The hardware interface (HWI) library is a collection of Simulink™ software interfaces for hardware utilized at the DLR in Bremen. Among others, this includes the interfaces for the attitude control hardware of the FACE. The library is subdivided into three sections that include the following interfaces:

- Generic

  Serial TX sends a series of bytes to a specified serial interface.

  Serial RX receives a series of bytes from a specified serial interface.

  TCP/IP Socket sends and receives a series of bytes over a TCP/IP capable device, acting as a client or server.

- Input/Sensors

  microFORSE-6U receives the sensor data from the corresponding LITEF optic rate gyro.

  AMR magnetometer ZARM receives the sensor data from the AMR magnetometer with a selectable frequency, and controls the power state of the device.

- Output/Actuators

  Astro- und Feinwerktechnik RW250 is the control interface for multiple RW 250 reaction wheels on one EIA-485 bus, and an interface to the housekeeping data.

  TMCM_310_stepper_control_over_IP controls the stepper velocities of the fine adjustment mechanisms of the assembly station, and also provides information about the end stop switches. The communication is tunneled through a TCP/IP connection.

HWI scripts

A set of MATLAB™ scripts is available for standard interactions with the OS of the OBC.

HWI_qnx.m executes an arbitrary shell command on the OBC.

HWI_shutdown.m shuts down the QNX operating system, but does not turn off the OBC itself.

HWI_slay.m kills the RTW application on the OBC in case it froze.

HWI_upload.m uploads the built real-time application to the OBC, executes it, and connects the *external mode* of Simulink™ to the application for live monitoring of the process.

Before these scripts can be used, two variables have to be initialized. These are the name of the model that should connect to the external mode, and the target IP addres of the OBC, for example:

```
HWI.ModelName='FACE_HWI_autoadjustment'
HWI.TargetIP='192.168.124.42'
```

7.5.2 FACE_lib

The *FACE_lib* is a collection of Simulink™ blocks that are commonly utilized in an ACS. Some include complex calculations, for example, Kalman filtered attitude determination, and some are only trivial implementations, such as the calculation of the control deviation between a guidance quaternion and the actual quaternion. Several reference implementations using these templates exist; they could be used as a starting point for a new ACS.

Gyro-based attitude determination

For the attitude determination a *configurable subsystem* is available that is implemented for inertial navigation with the μFORS-6U rate gyros. During a calibration period the mean angular rates are recorded to compensate for the Earth's rotation. It is possible to choose between two different KALMAN filter implementations (section 6.1) with different models for the linear system dynamics.

Linear control

Two LQR based controllers (section 6.2) can be used for simple attitude control. Additionally, the calculations for the control deviation based on the angular rate and on an attitude quaternion is done with a separate block.

Guidance

This library contains the implementation of a *slope limiter* (6.2.4) that calculates a trajectory towards a target attitude based on specific properties of the controlled system. The calculated slew maneuver starts with the target attitude of the previous maneuver, or the current measured attitude is used as a reference, depending on the chosen implementation.

Simple sequential control

The template *sequence* is a very basic process control instructing the ACS with new attitude quaternions after a specified condition is met. For example, it can test whether the attitude from the last step is reached before instructing the next target attitude, or it can check whether the stepper velocity has stayed below a given limited over a certain period of time before they are turned off.

Attitude calculations for ground station guidance

The automatic tracking of a ground station is calculated from different input parameters. These are the latitude and longitude of the ground station's location, the satellite position, and the instrument's line-of-sight that is to be aligned with the ground station. Additionally, a secondary condition is used to align another subsystem with a different target, e. g., align the solar panels with the Sun. The result is the guidance quaternion and the guidance angular rate.

7.6 Safe utilization of the FACE

During the initial tests and first experiments of the newly set up FACE, several problems occurred for which solutions were found, or at least ways to bypass critical situations.

The controller box from Fig. 7.2 is very sensitive towards power outages and drops in the supply hose for the compressed air. This is of course intentional, but the compressor

restarts just in time, barely above the critical pressure minimum. In a warm environment it could happen that the breakdown mode is then initiated, which deploys the pneumatic actuators and the currently running experiment has to be aborted. Once the breakdown mode is triggered, it is not possible to stop this action, but it can be prevented in the first place by using a well ventilated compressor casing.

At the beginning of an experiment, when the pneumatic actuators are still deployed, it should be ensured that the pressurized air from the compressor is not close to this minimum, since the retraction of the actuators could induce a large enough drop in the pressure system to trigger the breakdown mode. Also, in rare cases an experiment must be restarted because the retracting pneumatic actuators tear the assembly station down.

The user manual for the air bearing states the optimal thickness of the air gap between the semisphere and the bowl to be $(10.4 \pm 0.4)\ \mu\mathrm{m}$. This thickness is constantly measured and varies over time. Several times during experiments, the air cushion collapsed and the experiments had to be repeated. This can be avoided by using a larger gap. While this violates the manufacturer's guideline, it should only lead to a higher flow of compressed air through the bearing, and presumably to an increased friction. For very precise experiments, the compressor could be started early, reaching its operating temperature prior to the start of an experiment, and after recalibrating the air gap sensor and adjusting it to $10.4\,\mu\mathrm{m}$, the air cushion should not collapse under a constant load.

Prior to an experiment, the assembly station and all appendages must be checked to ensure that nothing can come into contact with the support stator. In addition, a wire or small pressure hose touching the assembly station can cause apparently inexplicable behavior during an experiment.

During the simulation of a satellite mission that is monitored with the control computer, it is possible to influence the ACS by using *switches* or *constants* from the Simulink™ library. Switches work fine, but changing a constant stops the simulation wile the pop-up window is visible. During this time, the external mode holds the 'real-time' application running on the OBC, and no controlling or measuring happens. This is not necessarily a big problem, but it has to be considered. Sadly, even a high work load of the control computer has the same effect, e. g. unlocking the Windows™ screen.

The rate gyros work in a free running mode, and during the time the OBC is halted by Simulink™ the sensor values are still received but not processed, which can lead to problems with the synchronization of the communication. This can result in wrong angular rates being calculated by the software interface. Through the use of a checksum, most of the errors can be intercepted, but with a $100\,\mathrm{Hz}$ sensor value signal using a one byte long checksum, it is not improbable for wrong a wrong sensor value with a seemingly correct checksum to appear once every few seconds. A similar situation arises at the beginning when the application was started on the OBC but still waits for the start command via the external mode. In these cases, the simulation should be repeated, especially as long as the rate gyros are used for inertial attitude determination.

In the general case of an unexpected behavior, aborting the experiment is the right choice. After stopping Simulink™, the reaction wheels could accelerate, based on the last instructed control torque, for another five seconds. When aborting a simulation in the state of a free floating satellite, possibly in motion, the pneumatic actuators have to

be deployed which results in high accelerations of the assembly station. In this situation it is suggested to manually slow down the satellite to avoid heavy loads in the bearings of the reaction wheels.

The laser pointer on the lower side of the assembly station can be used as a reference system in order to start all experiments in a similar attitude. The advantage is that differences in the course of an experiment are more obvious, and errors in the process sooner become apparent to the user. It is recommended to use an additional switch in the Simulink™ model for aborting a mission at any given time, steering the experimental satellite back to its home position with a robust controller.

For unattended experiments with long durations, an automatic error detection that is able to shut down the experimental satellite should be implemented, including for instance a slow desaturation of the reaction wheels to avoid long cycles of high spinning flywheels.

On a final note, it should be kept in mind that the wireless communication is not secure; this is a possible risk for a safe usage of the FACE.

8 Outlook

Currently the FACE is in an overall functioning state with some capabilities missing that will be brought into service as soon as the prerequisites are met. The magnetic simulator was set up and the HELMHOLTZ coils were tested with the manual interface and the control software. Measuring the magnetic field with magnetometers is possible but the more important utilization of the field for attitude control could not be tested. A control hardware for the magnetic torquers from section 3.2.3 is not available, therefore it is currently not possible to utilize the torquers as actuators for influencing the attitude of the satellite. Either the controller device must be newly developed, or a commercial device can be found that is able to drive the magnetic coils to interact with the externally generated magnetic field.

The assembly station is prepared for the fixation of the fluxgate magnetometer but no software interface has been developed yet, similar to the IMAR IMU, which could be integrated on the experimental satellite, but the OBC is missing the correct serial interface for receiving sensor data.

An important step towards a fully operational test facility is the relocation of the compressor necessary for the operation of the air bearing. Currently, the working compressor induces oscillations on the test facility that are measured by the fiber optic gyros, indicating a negative influence on the experiment. The climatisation of the laboratory is another important area that needs to be addressed. A change in temperature is likely to influence the calibrated c. m. of the assembly station. Perhaps the relocation of the heat generating compressor suffices to solve the problem, if not, an air-conditioned laboratory environment must be considered.

The fine adjustment mechanisms only work in a limited velocity range. If they are commanded a higher velocity, the stepper motors start losing steps and eventually come to a halt. For a fine adjustment process that is capable of aligning the c. m. from a coarser starting point, higher velocities are mandatory and therefore the mechanisms must be improved.

During the experiments of the ground station guidance mission it became apparent that a visualization of this mission would be very helpful. Combining an experiment on the assembly station and a numerical simulation with a real-time preview of the overall simulation should simplify the implementation process and the troubleshooting. In the ground station example, such a visualization could show the pass of the satellite and the necessary reference frames, ensuring a correct test procedure through comparison with the attitude of the assembly station.

9 Summary

During the course of this diploma thesis several topics from different engineering areas have been discussed and worked on. The basics for the necessary mathematical description of the celestial mechanics are introduced and the fundamentals for the attitude dynamics of a spacecraft are described in detail.

In the course of setting up the overall test facility, every component has been documented with its primary features and specialties, including the hardware for the experimental satellite. This hardware is then put into operation separately. The wiring of the devices are carefully conducted according to the corresponding documentation, followed by first tests of the communication. Even though the protocol of the fiber optic gyroscopes is very simple, implementing a robust communication led to complications. However, the final version of the software interface is capable of resynchronizing to the steady stream of sensor values, while still maintaining a memory efficient implementation.

The communication with the reaction wheels caused some problems. It took a week before the first acknowledgement packet from the wheels had been received by the OBC. Monitoring the transmissions with an oscilloscope and testing all permutations of communication lines and their redundant counterparts did not result in a successful exchange of information, even though the power consumption of the device clearly indicated an operational state. In the end, it turned out that the technical documentation states a protocol which is not implemented on the reaction wheels. The utilization of a different protocol works flawlessly and the implemented software interface can efficiently handle multiple devices simultaneously.

The control board for the fine adjustment steppers was already working with a provided software. The additional usage of the steppers from the Simulink™ environment has been requested to use exactly the same protocol. For this purpose the communication between the stepper board and the provided software is tunneled through a self-implemented software, acquiring the correct properties of the employed protocol in order to implement the new interface accordingly.

Using the software interfaces implemented for the utilization in the MATLAB™ extension RTW, and the integrated hardware on board of the experimental satellite, first tests could be conducted, proving the ability of the OBC, the actuators, and the sensors to work together. For the automatic controlling of the satellite an attitude determination is necessary. The optic gyros are used for an inertial estimation of the attitude and an available Kalman filter implementation is modified for the dynamics of a spacecraft to gain smoother sensor signals. These signals also have to be compensated for the Earth's rotation that is measured by high precision rate gyros.

For the design of PID controllers the very robust and widespread LQR approach is employed. The resulting controllers are only capable of efficiently decreasing the deviation

to a guidance in a small surrounding of a constant operation point. For a better guidance accuracy a feed forward control is implemented in order to take the non-linear dynamics of the satellite's motion into account.

The developed ACS is used in a very robust form to control two processes during the initial operations. The process of self-aligning the c. m. uses the gravitational acceleration as an indication in order to control the fine adjustment mechanisms, reducing the disturbance torque caused by the off-centered c. m. The second process that relies on the robust ACS is the automatic sequence cycling through different attitudes, recording the measurement data for later evaluation in order to estimate the satellite's moment of inertia.

This estimate and the aligned c. m. are important for simulating the demonstration missions. A long sequence of attitudes is used for a guidance trajectory that has to be followed by the experimental satellite. The ACS works very well, although the step in the feed forward control torque induces an oscillation every time the next attitude is targeted. A different algorithm for the open-loop control with a smoother evolution of the torque could increase the overall guidance accuracy and even better results can be expected.

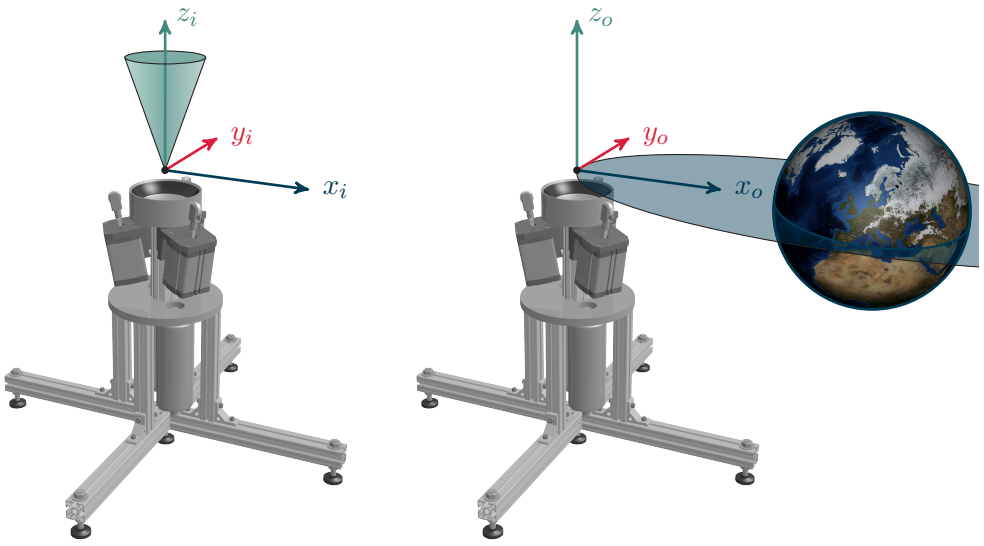
The ground station guidance mission shows the accuracy limit of what is possible with the currently utilized hardware. Even though the absolute accuracy limit may be reached, the time to reach a steady state could be lowered by a more advanced controller and guidance design. An indication for this limit is the random walk of the rate gyros that is visible in the course of the deviation from the guidance. An instrument with an absolute reference could have the advantage of a better attitude determination without the random drift in order to specify the deviation from the guidance to an external reference, not to the inertially measured reference.

The result of this thesis is a well documented implementation of a basic ACS, which can be used as a starting point for new developments or extensions of the old one. In the course of this diploma thesis it is carefully taken care of a correct description of all used coordinate systems and their alignment towards each other. The equations and models from the standard literature are employed as close as possible to the original descriptions and at the same time coherent with the remainder of this thesis' nomenclature.

A Reference systems

A.1 Laboratory system

Experiments in the FACE require a reference system to describe the attitude of the air bearing table with respect to the support stator. The z axis is defined to point upwards opposing the gravitational direction. The x and y axes are defined to complete an orthonormal basis, shown in both images of Fig. A.1. This definition leaves the freedom to



(a) Pseudo inertial frame (b) Pseudo inertial frame aligned with an orbit plane

Figure A.1: Laboratory reference systems

rotate around the z axis to the user's demands, which at the same time has no influence on the maximal deflection angles of the assembly station with respect to the laboratory's frame. This is indicated by the cone in the left picture, which is symmetric around the vertical axis. The cone angle is $20°$, following the design specification of the air bearing table. The origin of this frame is the pivot point of the air bearing.

The first figure shows the common reference system for an assumed inertial frame i, while the second figure is a special frame o for tracking a ground station from orbit, leaving most of the rotation to the z_o axis.

A.2 Body fixed satellite system

The body fixed reference system for the experimental satellite is depicted in Fig. A.2. So

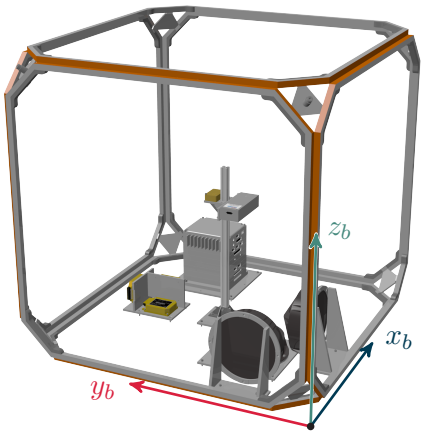


Figure A.2: Body fixed reference frame of the experimental satellite

far it has not been not necessary to specify the location of the frame's origin. The attitude on the other hand is defined with respect to the laboratory's frame from section A.1. At the beginning of an experiment when the pneumatic actuators hold the assembly station in position, all axes of the body fixed frame and the laboratory frame coincide. This is a definition resulting from the inertially determined attitude that requires a reference attitude. The major axes of this frame are each aligned with a rate gyro's sensitive axis, and a reaction wheel's sensitive axis.

A.3 Assembly station system

The assembly station's frame is defined through the alignment of the fine adjustment mechanisms shown in Fig. A.3. This definition does not specify the origin of this frame, as the fine adjustment axes do not intersect in a single location. Similar to the body fixed's origin the definition of a location was not necessary, only the attitude towards other frames.

A.4 Earth centered inertial (ECI)

The ECI i frame is located at the Earth's center with its positive z axis aligned with the Earth's rotation axis. The x axis points towards the *vernal equinox*, a reference direction defined by the Sun's location relative to the Earth's location on a specific date. The y axis is perpendicular to both other axes and completes the basis.

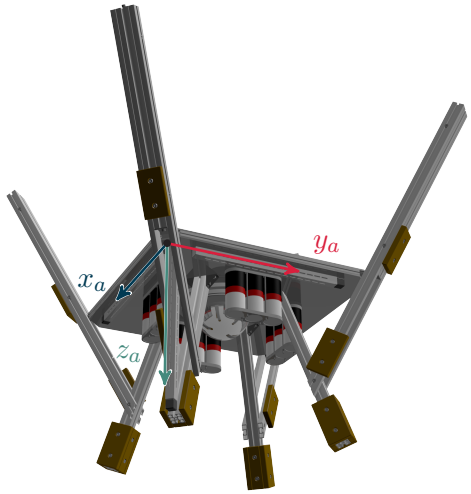


Figure A.3: Assembly station reference frame

A.5 Earth centered earth fixed (ECEF)

The ECEF e frame is fixed to the rotating Earth. The z axis, coinciding with the z axis of the ECI frame, points in the direction of the Earth's rotation axis. The x axis points towards the *prime meridian* (located in Greenwich, United Kingdom), and the y axis completes the perpendicular basis.

B German summary

Das DLR-Institut für Raumfahrtsysteme mit Sitz in Bremen hat ein neues Labor in Auftrag gegeben, welches für die Erforschung und die experimentelle Verifikation neuer Ansätze in der Lageregelung gedacht ist. Das Labor mit seiner speziellen Einrichtung wurde *Facility for Attitude Control Experiments (FACE)* benannt. Der zentrale Teststand ist ein Luftlagertisch und wurde von der Firma *Astro- und Feinwerktechnik Adlershof GmbH* gefertigt. Das Luftlager besteht aus einer festen Schale in der eine Halbkugel liegt. Zwischen beiden Elementen wird mit Druckluft ein Luftkissen aufgebaut, welches es ermöglicht die Halbkugel sehr reibungsarm zu bewegen. An dieser Halbkugel ist eine Tragplattform befestigt auf der Experimente zur Lageregelung montiert werden können. Diese Plattform kann sich beliebig weit um die Hochachse drehen, ist in den beiden anderen körperfesten Achsen aber begrenzt durch mechanische Anschläge bei Auslenkungswinkeln von ungefähr $20°$. An der Plattform ist eine akkubetriebene Stromversorgung angebracht, die die Hardware für einen experimentellen Satelliten für mehrere Stunden betreiben kann.

Um den Luftlagertisch herum sind weitere Einrichtungen aufgebaut, die für die Simulation der Weltraumumgebung gedacht sind. Ein Aufbau von sechs Helmholtz-Spulen dient der präzisen Nachbildung des Erdmagnetfeldes und ein Studioscheinwerfer wird genutzt, um die Beleuchtung durch die Sonne im begrenzten Maße nachzuahmen.

Das Ziel dieser Diplomarbeit ist es dieses Labor aufzubauen und die zur Verfügung stehende Hardware in Betrieb zu nehmen. Dazu gehören auch die Geräte für einen experimentellen Satelliten, die auf die Tragplattform integriert werden. Zum Abschluss der Arbeit sollen die Fähigkeiten der Einrichtung anhand von Beispielmissionen demonstriert werden, die eine realistische Simulation von Manövern aus der Raumfahrt beinhalten. Hierzu sind am Anfang der Arbeit die Grundlagen zusammengestellt, die für die Orbit- und die Starrkörperdynamik eines frei fliegenden Satelliten benötigt werden.

Im Anschluss sind die Hauptkomponenten des Labors einzeln beschrieben, die im Laufe der Arbeit in Betrieb genommen wurden. Die Geräte für den experimentellen Satelliten werden anhand der technischen Dokumentation kurz vorgestellt, sowie deren Funktionsweisen kurz erläutert. Danach folgt die Inbetriebnahme dieser Geräte. Die derzeitige funktionsfähige Ausstattung umfasst faseroptische Kreisel für die Lagebestimmung, Reaktionsschwungräder für die Lageregelung und einen On-board Computer für die Kommunikation zwischen den Geräten, sowie die Auswertung der Sensordaten. Ein weiteres Gerät ist zur Ansteuerung von Lineareinheiten gedacht, die drei $100\,\mathrm{g}$-Gewichte entlang der drei körperfesten Achsen der Tragplattorm verschieben können, um den Schwerpunkt der Tragplattform und der Satellitenhardware zu beeinflussen.

Jedes der genannten Geräte wird einzeln in der Laborumgebung an ein Labornetzteil angeschlossen und getestet. Nachdem die serielle Kommunikation erfolgreich getestet ist, wird eine Softwareschnittstelle definiert, die für den Betrieb in der MATLAB™/Si-

mulink™ Entwicklungsumgebung ausgelegt ist. Dies hat den entscheidenden Vorteil, dass eine sehr einfache Ansteuerung der Satellitenhardware möglich ist ohne das der Entwickler genauere Kenntnisse besitzen muss, wie die serielle Kommunikation abläuft. Letztendlich kann ein in Simulink™ entworfenes Lageregelungssystem mit der Zusatzsoftware Real-Time Workshop™ in eine Echtzeitanwendung kompiliert werden und auf dem On-board Computer ausgeführt werden.

Die Implementierung dieser Schnittstellen orientiert sich an den Protokollen, wie sie in den entsprechenden Dokumentationen für die Geräte beschrieben sind. Eine Besonderheit der optischen Kreisel ist, dass sie ungefragt Sensordaten über die serielle Leitung schicken, die dann vom On-board Comuter empfangen und entschlüsselt werden. Hierbei kann es zu Synchronisationsproblemen kommen, bei dem der Computer in dem konstanten Datenstrom austesten muss wo ein Datenpaket anfängt und wo es aufhört. Dies führte bei späteren Versuchen zu Problemen, bei denen die synchrone Kommunikation nicht mehr korrekt aufrechterhalten werden konnte. Die endgültige Software ist nun in der Lage in einer solchen Situation die Kommunikation zu resynchronisieren, wobei allerdings einige Datenpakete verloren gehen können, was zu Problemen bei der inertialen Lagebestimmung führen kann.

Die Versuche die Reaktionsschwungräder in Betrieb zu nehmen blieben anfangs ohne Erfolg. Alle zur Verfügung stehenden Werkzeuge wurden verwendet, um eine ordnungsgemäße Funktion der seriellen Kommunikation zu bestätigen. Letztendlich war es eine fehlerhafte Anleitung vom Hersteller, die ein falsches Protokoll dokumentierte, welches gar nicht auf den Rädern implementiert wurde.

Für die Ansteuerung der Lineareinheiten zur Feineinstellung des Schwerpunktes ist eine Softwareschnittstelle notwendig, die die Kommunikation über WLAN beherrscht. Die Feinverstellungen wurde vom Hersteller dafür ausgelegt von einem Kontrollcomputer im Labor kabellos verwendet zu werden, um manuell den Schwerpunkt so zu verschieben, dass er mit Drehpunkt des Luftlagers übereinstimmt. Die direkte Kontrolle vom On-board Computer war nicht vorgesehen, durch die gegebene Schnittstelle allerdings möglich, sodass der Schwerpunkt nun auch direkt vom Lageregelungssystem des experimentellen Satelliten beeinflusst werden kann.

Nachdem die Softwareschnittstellen zu der Satellitenhardware fertig gestellt wurden, konnte die Integration der Aktuatoren und Sensoren auf die Tragplattform durchgeführt werden. Die Kabel für die Kommunikation und die Stromversorgung wurden sorgfältig verlegt, um ein verrutschen bei der Durchführung von Experimenten zu verhindern. Der darauf folgen Schritt ist die Entwicklung eines Lageregelungssystems, welches den Satelliten einer vorgegebenen Trajektorie folgen lässt, indem es die Lage im Raum mit den Kreiseln misst und mit den Rädern beeinflusst.

Die Lagebestimmung beruht derzeitig auf der inertialen Messung der Drehraten und deren Integration. Für die Rauschunterdrückung wird eine KALMAN-Filter-Implementierung genutzt, die auf unterschiedlichen mathematischen Modellen der Satellitendynamik basiert. Zusätzlich muss die Erddrehrate berücksichtigt werden, welche eine Drift in der Lagebestimmung als Folge hat, wenn sie nicht kompensiert wird.

Für die Lageregelung werden lineare PID-Regler verwendet, die mit dem weitverbreiteten LQR-Verfahren (linear quadratic regulator) ausgelegt werden. Diese Regler sind im All-

gemeinen sehr robust, allerdings liefern sie auch nur gute Ergebnisse in einem begrenzten Umfeld um einen fest definierten Arbeitspunkt. Zur Verbesserung der Führungsgenauigkeit sind zwei Vorsteuerungen implementiert. Das LQR-Verfahren basiert auf der linearisierten Dynamik des zu regelnden Systems, die nichtlinearen Terme der Dynamik werden in Echtzeit berechnet und können als Störgrößenaufschaltung die Regelung unterstützen. Die zweite Vorsteuerung ist eine geplante Trajektorie zwischen Soll- und Istlage mit der das Regelmoment für die Reaktionsschwungräder als Führungsgröße vorgegeben werden kann, wodurch die Linearregler entlastet werden. Beide Vorsteuerungen benötigen für gute Führungsgenauigkeiten eine gute Schätzung des Trägheitsmomentes vom Gesamtsatelliten (Tragplattform und Satellitenhardware).

Die erste Aufgabe für das neu entwickelte Lageregelungssystem ist die automatische Korrektur des Schwerpunktes, sodass Schwer- und Drehpunkt des Luftlagers ineinander fallen. Dies ist notwendig um einen kräftefreien experimentellen Satelliten simulieren zu können, welcher sonst von einem Störmoment durch die Gravitationsbeschleunigung beeinträchtigt würde. Die Gewichte in den Lineareinheiten werden derart angesteuert, dass die Reaktionsschwungräder im Verlauf des Vorgangs immer weniger Regelmomente liefern müssen. Als Indiz für einen gut eingestellten Schwerpunkt gilt, wenn der Satellit in einer beliebigen Lage fast ohne Regelmomente auskommt, um seine Lage zu halten.

Der austarierte Satellit wird nun vom Lageregelungssystem mit einer schnellen Abfolge von wechselnden Zielausrichtungen kommandiert. Die Messdaten der Sensoren werden aufgezeichnet, um sie in einem späteren Verfahren auszuwerten. Das Ziel dieser Messung ist es, das Trägheitsmoment abzuschätzen, welches Einfluss auf das gesamte Lageregelungssystem hat. Erst durch ein genau bestimmtes Trägheitsmoment kann die Vorsteuerung gute Werte liefern, die Regler gut ausgelegt werden und die Kalman-Filter eine gute Rauschunterdrückung erzielen.

Die Demonstration der Funktion des Satelliten und der Funktion des gesamten Labors beinhaltet zwei Satellitenmissionen, die in mehreren Experimenten simuliert werden. Mit einer Abfolge von unterschiedlichsten Zielausrichtungen ist eine Trajektorie bestimmt worden, der der Satellit so genau wie möglich folgen soll. Der Grad der Abweichungen von dieser Trajektorie hängen deutlich mit einem Sprung in der Vorsteuerung zusammen. Diese Vermutung wurde durch die selbe Mission mit langsameren Manövern bestätigt, in der geringere Regelmomenten kommandiert werden und damit der Sprung kleiner ausfällt. Neben der Möglichkeit längere Missionen durchzuführen verdeutlicht diese Mission außerdem, dass es gut möglich ist Versuche zu wiederholen und mit vorangegangenen zu vergleichen.

Eine zweite Demonstrationsmission zeigt die Simulation des Überfluges eines Satelliten über eine Bodenstation, die mit einem On-board Instrument verfolgt wird. Während der Verfolgungsphase wurden sehr gute Führungsgenauigkeiten demonstriert, die wahrscheinlich auch die ungefähren Grenzen des zurzeit Machbaren aufzeigen. Ohne eine Verbesserung der Lagebestimmung hat der *Random Walk* der faseroptischen Kreisel bei einer Führungsgenauigkeit im Bereich von einigen hundertstel Grad einen deutlich sichtbaren Einfluss. Eine externe Referenz zur Lagebestimmung ohne dieses genannte Verhalten hätte zudem den Vorteil, dass eine absolute Genauigkeit der Regelung angegeben werden kann, und nicht nur die Genauigkeit zu der inertial gemessenen Lage.

Die Diplomarbeit schließt mit einem Kapitel, das die Prozedur beschreibt, wie neue Hardware oder neue Algorithmen auf dem Teststand integriert, beziehungsweise implementiert werden können. Es beinhaltet die Erfahrungen, die während des Aufbaus des Labors gewonnen wurden, die nützlich seien sollten, um effizient und sicher im FACE zu arbeiten.

C Software interfaces

C.1 Astro- und Feinwerktechnik Adlershof reaction wheel RW 250

C.1.1 Simulink™ block

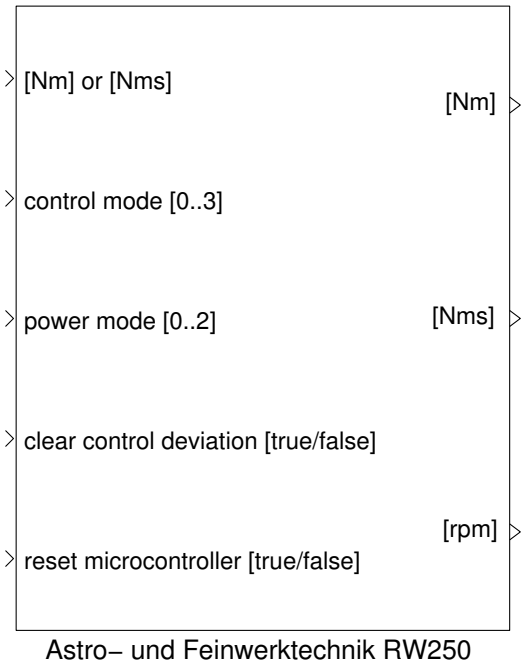


Figure C.1: Simulink™ block for the RW 250 software interface

C.1.2 Source code

Listing C.1: HWI_RW250_sf.c

```
1  #define S_FUNCTION_NAME  HWI_RW250_sf
2  #define S_FUNCTION_LEVEL 2
3
4  /*
5   * Need to include simstruc.h for the definition of the SimStruct and
6   * its associated macro definitions.
7   */
8  #include "simstruc.h"
9
```

```
10 #ifndef MATLAB_MEX_FILE
11 // include files here
12 //#include <time.h>
13 #include <sys/select.h> //for timeval
14 #endif
15
16 // DEBUGMODE ----------------------
17 //#define DEBUG 1
18 // ----------------------------------
19
20 #define RWADDRESSES (ssGetSFcnParam(S, 0))   /* Parameter MOTORADDRESS */
21 #define MAXRXBYTES 60                        /* Maximal RXbytes per cycle */
22 #define MAXTXBYTES 60
23 #define MAX_REPLY_WAIT 1 //Wait maximal x seconds for a requested packet.
24 #define TIME_BETWEEN_CONTROL_COMMANDS 0.5
25 //#define PACKETBUFFERSIZE 60;
26
27 // Packet definitions:
28 #define STX_command 0x23
29 #define STX_reply 0x3c
30 #define ADR_master 0x40
31 #define ADR_universal 0x77
32 #define EOX 0x0d
33
34 //Command definitions
35 #define COM_SET_OMEGA_SIMPLE 0x6a
36 #define COM_SET_OMEGA_STRAT 0x65
37 #define COM_SET_DEOMEGA_COARSE 0x63
38 #define COM_SET_DEOMEGA_ADAPT 0x69
39 #define COM_SET_POWER 0x6c
40 #define COM_CLEAR_DELTA_TH 0x66
41 #define COM_RESTART 0x72
42 #define COM_SENDHK 0x74
43 #define COM_SENDDATA 0x6f
44
45 //MODES
46 #define MODE_SET_OMEGA_SIMPLE 0
47 #define MODE_SET_OMEGA_STRAT 1
48 #define MODE_SET_DEOMEGA_COARSE 2
49 #define MODE_SET_DEOMEGA_ADAPT 3
50
51
52 // DO_COMMAND
53 #define DO_COMMAND_CONTROL_VALUE 0
54 #define DO_COMMAND_SET_POWER 1
55 #define DO_COMMAND_CLEAR_DELTA_TH 2
56 #define DO_COMMAND_RESTART 3
57 #define DO_COMMAND_SENDHK 4
58 #define DO_COMMAND_SENDDATA 5
59
60 static void mdlInitializeSizes(SimStruct *S)
61 {
62     // Standard port width is the number of reaction wheels on the bus.
63     // Size is determined by the number of RWADDRESSES
64     uint8_T pwidth = (uint8_T) mxGetNumberOfElements(RWADDRESSES);
65     // Parameters from the mask
66     ssSetNumSFcnParams(S, 1);  /* Number of expected parameters: Vector of RWADRESSES
           */
67     if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) {
68         /* Return if number of expected != number of actual parameters */
69         return;
70     }
```

```
71
72      //-------------------------------------------------------------------------//
73      // Input ports: 9
74      if (!ssSetNumInputPorts(S, 9)) return;
75      /*Input Port 0: control rpm or rpm/s */
76      ssSetInputPortWidth(S, 0,pwidth);
77      ssSetInputPortDataType(S, 0, SS_DOUBLE);
78      ssSetInputPortComplexSignal(S,  0, COMPLEX_NO);
79      ssSetInputPortRequiredContiguous(S, 0, true); /*direct input signal access*/
80      ssSetInputPortDirectFeedThrough(S, 0, 1);
81      /*Input Port 1: control mode */
82      ssSetInputPortWidth(S, 1,pwidth);
83      ssSetInputPortDataType(S, 1, SS_UINT8);
84      ssSetInputPortComplexSignal(S,  1, COMPLEX_NO);
85      ssSetInputPortRequiredContiguous(S, 1, true); /*direct input signal access*/
86      ssSetInputPortDirectFeedThrough(S, 1, 1);
87      /*Input Port 2: set power mode */
88      ssSetInputPortWidth(S, 2,pwidth);
89      ssSetInputPortDataType(S, 2, SS_UINT8);
90      ssSetInputPortComplexSignal(S,  2, COMPLEX_NO);
91      ssSetInputPortRequiredContiguous(S, 2, true); /*direct input signal access*/
92      ssSetInputPortDirectFeedThrough(S, 2, 1);
93      /*Input Port 3: clear control deviation */
94      ssSetInputPortWidth(S, 3,pwidth);
95      ssSetInputPortDataType(S, 3, SS_BOOLEAN);
96      ssSetInputPortComplexSignal(S,  3, COMPLEX_NO);
97      ssSetInputPortRequiredContiguous(S, 3, true); /*direct input signal access*/
98      ssSetInputPortDirectFeedThrough(S, 3, 1);
99      /*Input Port 4: RXBytes */
100     ssSetInputPortWidth(S, 4, MAXRXBYTES );
101     ssSetInputPortDataType(S, 4, SS_UINT8);
102     ssSetInputPortComplexSignal(S,  4, COMPLEX_NO);
103     ssSetInputPortRequiredContiguous(S, 4, true); /*direct input signal access*/
104     ssSetInputPortDirectFeedThrough(S, 4, 1);
105     /*Input Port 5: getNumRXBytes */
106     ssSetInputPortWidth(S, 5,1);
107     ssSetInputPortDataType(S, 5, SS_UINT32);
108     ssSetInputPortComplexSignal(S,  5, COMPLEX_NO);
109     ssSetInputPortRequiredContiguous(S, 5, true); /*direct input signal access*/
110     ssSetInputPortDirectFeedThrough(S, 5, 1);
111     /*Input Port 6: RXStatus */
112     ssSetInputPortWidth(S, 6,1);
113     ssSetInputPortDataType(S, 6, SS_INT32);
114     ssSetInputPortComplexSignal(S,  6, COMPLEX_NO);
115     ssSetInputPortRequiredContiguous(S, 6, true); /*direct input signal access*/
116     ssSetInputPortDirectFeedThrough(S, 6, 1);
117     /*Input Port 7: getNumTXBytes */
118     ssSetInputPortWidth(S, 7,1);
119     ssSetInputPortDataType(S, 7, SS_UINT32);
120     ssSetInputPortComplexSignal(S,  7, COMPLEX_NO);
121     ssSetInputPortRequiredContiguous(S, 7, true); /*direct input signal access*/
122     ssSetInputPortDirectFeedThrough(S, 7, 1);
123     /*Input Port 8: restart microcontroller */
124     ssSetInputPortWidth(S, 8,pwidth);
125     ssSetInputPortDataType(S, 8, SS_BOOLEAN);
126     ssSetInputPortComplexSignal(S,  8, COMPLEX_NO);
127     ssSetInputPortRequiredContiguous(S, 8, true); /*direct input signal access*/
128     ssSetInputPortDirectFeedThrough(S, 8, 1);
129     //-------------------------------------------------------------------------//
130     //-------------------------------------------------------------------------//
131     // Output ports ports: 17
132     if (!ssSetNumOutputPorts(S, 17)) return;
```

```
133     /* Output Port 0: OmegaSoll (OmegaDemanded in 0.12/min) */
134     ssSetOutputPortWidth(S, 0, pwidth);
135     ssSetOutputPortDataType(S, 0, SS_DOUBLE);
136     ssSetOutputPortComplexSignal(S, 0, COMPLEX_NO);
137     /* Output Port 1: DeTheta (Control deviation in 360°/2000) */
138     ssSetOutputPortWidth(S, 1, pwidth);
139     ssSetOutputPortDataType(S, 1, SS_DOUBLE);
140     ssSetOutputPortComplexSignal(S, 1, COMPLEX_NO);
141     /* Output Port 2: LimLow (lower (negativ) acceleration limit in 0.008/s^2) */
142     ssSetOutputPortWidth(S, 2, pwidth);
143     ssSetOutputPortDataType(S, 2, SS_DOUBLE);
144     ssSetOutputPortComplexSignal(S, 2, COMPLEX_NO);
145     /* Output Port 3: LimHigh (higher (positiv) acceleration limit in 0.008/s^2) */
146     ssSetOutputPortWidth(S, 3, pwidth);
147     ssSetOutputPortDataType(S, 3, SS_DOUBLE);
148     ssSetOutputPortComplexSignal(S, 3, COMPLEX_NO);
149     /* Output Port 4: OmegaEstimated (estimated rate after a 125ms interval in 0.12/min
            ) */
150     ssSetOutputPortWidth(S, 4, pwidth);
151     ssSetOutputPortDataType(S, 4, SS_DOUBLE);
152     ssSetOutputPortComplexSignal(S, 4, COMPLEX_NO);
153     /* Output Port 5: Voltage ( in 0.01V) */
154     ssSetOutputPortWidth(S, 5, pwidth);
155     ssSetOutputPortDataType(S, 5, SS_DOUBLE);
156     ssSetOutputPortComplexSignal(S, 5, COMPLEX_NO);
157     /* Output Port 6: Current ( in 0.01A) */
158     ssSetOutputPortWidth(S, 6, pwidth);
159     ssSetOutputPortDataType(S, 6, SS_DOUBLE);
160     ssSetOutputPortComplexSignal(S, 6, COMPLEX_NO);
161     /* Output Port 7: Temprature motor ( in 0.1°C) */
162     ssSetOutputPortWidth(S, 7, pwidth);
163     ssSetOutputPortDataType(S, 7, SS_DOUBLE);
164     ssSetOutputPortComplexSignal(S, 7, COMPLEX_NO);
165     /* Output Port 8: Temprature electronic ( in 0.1°C) */
166     ssSetOutputPortWidth(S, 8, pwidth);
167     ssSetOutputPortDataType(S, 8, SS_DOUBLE);
168     ssSetOutputPortComplexSignal(S, 8, COMPLEX_NO);
169     /* Output Port 9: Status bits */
170     ssSetOutputPortWidth(S, 9, pwidth);
171     ssSetOutputPortDataType(S, 9, SS_UINT16);
172     ssSetOutputPortComplexSignal(S, 9, COMPLEX_NO);
173     /* Output Port 10: Power index  */
174     ssSetOutputPortWidth(S, 10, pwidth);
175     ssSetOutputPortDataType(S, 10, SS_UINT8);
176     ssSetOutputPortComplexSignal(S, 10, COMPLEX_NO);
177     /* Output Port 11: estimated friction Z1 */
178     ssSetOutputPortWidth(S, 11, pwidth);
179     ssSetOutputPortDataType(S, 11, SS_DOUBLE);
180     ssSetOutputPortComplexSignal(S, 11, COMPLEX_NO);
181     /* Output Port 12: Omega measured (in 0.12/min) */
182     ssSetOutputPortWidth(S, 12, pwidth);
183     ssSetOutputPortDataType(S, 12, SS_DOUBLE);
184     ssSetOutputPortComplexSignal(S, 12, COMPLEX_NO);
185     /* Output Port 13: TXBytes */
186     ssSetOutputPortWidth(S, 13, MAXTXBYTES);
187     ssSetOutputPortDataType(S, 13, SS_UINT8);
188     ssSetOutputPortComplexSignal(S, 13, COMPLEX_NO);
189     /* Output Port 14: setNumTXBytes */
190     ssSetOutputPortWidth(S, 14, 1);
191     ssSetOutputPortDataType(S, 14, SS_UINT32);
192     ssSetOutputPortComplexSignal(S, 14, COMPLEX_NO);
193     /* Output Port 15: setNumRXBytes */
```

```
194     ssSetOutputPortWidth(S, 15, 1);
195     ssSetOutputPortDataType(S, 15, SS_UINT32);
196     ssSetOutputPortComplexSignal(S, 15, COMPLEX_NO);
197     /* Output Port 16: deltaomega */
198     ssSetOutputPortWidth(S, 16 , pwidth);
199     ssSetOutputPortDataType(S, 16, SS_DOUBLE);
200     ssSetOutputPortComplexSignal(S, 16, COMPLEX_NO);
201     // global variables:
202     ssSetNumIWork(S, 8); //integer: modus, rxbuffer_position,num_rw,rw_cycle,
            command_cycle,command_cycle_mem,packet_reply_watchdog,start derivatives,
203     ssSetNumPWork(S, 4); // pointer:rxbuffer,ticktock,deltaclock,lastomega
204 }
205
206 /* Function: mdlInitializeSampleTimes =========================================
207  * Abstract:
208  *    This function is used to specify the sample time(s) for your
209  *    S-function. You must register the same number of sample times as
210  *    specified in ssSetNumSampleTimes.
211  */
212 static void mdlInitializeSampleTimes(SimStruct *S)
213 {
214     ssSetSampleTime(S, 0, INHERITED_SAMPLE_TIME);
215     ssSetOffsetTime(S, 0, FIXED_IN_MINOR_STEP_OFFSET);
216
217 }
218
219
220
221 #define MDL_INITIALIZE_CONDITIONS   /* Change to #undef to remove function */
222 #if defined(MDL_INITIALIZE_CONDITIONS)
223   /* Function: mdlInitializeConditions ========================================
224    * Abstract:
225    *    In this function, you should initialize the continuous and discrete
226    *    states for your S-function block.  The initial states are placed
227    *    in the state vector, ssGetContStates(S) or ssGetRealDiscStates(S).
228    *    You can also perform any other initialization activities that your
229    *    S-function may require. Note, this routine will be called at the
230    *    start of simulation and if it is present in an enabled subsystem
231    *    configured to reset states, it will be call when the enabled subsystem
232    *    restarts execution to reset the states.
233    */
234   static void mdlInitializeConditions(SimStruct *S)
235   {
236   }
237 #endif /* MDL_INITIALIZE_CONDITIONS */
238
239 #define SET_RXBUFFER(a) ssSetPWorkValue(S,0,a)
240 #define RXBUFFER ssGetPWorkValue(S,0)
241
242 #define SET_TICKTOCK(a) ssSetPWorkValue(S,1,a)
243 #define TICKTOCK ssGetPWorkValue(S,1)
244
245 #define SET_DELTA_CLOCK(a) ssSetPWorkValue(S,2,a)
246 #define DELTA_CLOCK ssGetPWorkValue(S,2)
247
248 #define SET_LAST_OMEGA(a) ssSetPWorkValue(S,3,a)
249 #define LAST_OMEGA ssGetPWorkValue(S,3)
250
251 #define SET_MODUS(a) ssSetIWorkValue(S,0,a)
252 #define MODUS ssGetIWorkValue(S,0)
253
254 #define SET_RXBUFFER_POSITION(a) ssSetIWorkValue(S,1,a)
```

```
255 #define RXBUFFER_POSITION ssGetIWorkValue(S,1)
256
257 #define SET_NUM_RW(a) ssSetIWorkValue(S,2,a)
258 #define NUM_RW ssGetIWorkValue(S,2)
259
260 #define SET_RW_CYCLE(a) ssSetIWorkValue(S,3,a)
261 #define RW_CYCLE ssGetIWorkValue(S,3)
262
263 #define SET_COMMAND_CYCLE(a) ssSetIWorkValue(S,4,a)
264 #define COMMAND_CYCLE ssGetIWorkValue(S,4)
265
266 #define SET_COMMAND_CYCLE_MEM(a) ssSetIWorkValue(S,5,a)
267 #define COMMAND_CYCLE_MEM ssGetIWorkValue(S,5)
268
269 #define SET_PACKET_REPLY_WATCHDOG(a) ssSetIWorkValue(S,6,a)
270 #define PACKET_REPLY_WATCHDOG ssGetIWorkValue(S,6)
271
272 #define SET_START_DERIVATIVES(a) ssSetIWorkValue(S,6,7)
273 #define START_DERIVATIVES ssGetIWorkValue(S,7)
274
275 #define MDL_START  /* Change to #undef to remove function */
276 #if defined(MDL_START)
277   /* Function: mdlStart =======================================================
278    * Abstract:
279    *    This function is called once at start of model execution. If you
280    *    have states that should be initialized once, this is the place
281    *    to do it.
282    */
283   static void mdlStart(SimStruct *S)
284   {
285     #ifndef MATLAB_MEX_FILE
286     int i=0;
287     // Array of chars for the receiving buffer. If not all data for one packet is
288     // received in on function call the data is saved to be used when the rest arrives.
289     char * rxbuffer=malloc(sizeof(char)*MAXRXBYTES);
290     // clear buffer
291     bzero(rxbuffer,MAXRXBYTES);
292     // Make pointer to other functions visible.
293     SET_RXBUFFER(rxbuffer);
294     // Initialize variables
295     SET_RXBUFFER_POSITION(0);
296     SET_NUM_RW((uint32_T) mxGetNumberOfElements(RWADDRESSES));
297     SET_RW_CYCLE(0);
298     SET_MODUS(0); // Set send modus
299     // Time between commands to reaction wheel must be at least .5 seconds.
300     int * ticktock=malloc(sizeof(int)*NUM_RW);
301     for(i=0;i<NUM_RW;i++)
302     {
303         ticktock[i]=0;
304     }
305     SET_TICKTOCK(ticktock);
306     // For calculating the derivative of the angular rate OMEGA better
307     // Time measuring is used. There won't be big differences to the ticktock way.
308     struct timeval * deltaclock=malloc(sizeof(struct timeval)*NUM_RW);
309     SET_DELTA_CLOCK(deltaclock);
310
311     struct timeval * deltaclocktmp=DELTA_CLOCK;
312     for(i=0;i<NUM_RW;i++)
313     {
314         gettimeofday(&deltaclocktmp[i],0); //wrong but first value does not matter
315     }
316     // Save last OMEGAS for derivative calculations
```

```
317     double * lastomega=malloc(sizeof(double)*NUM_RW);
318     SET_LAST_OMEGA(lastomega);
319     for(i=0;i<NUM_RW;i++)
320     {
321         lastomega[i]=0.;
322     }
323     //start with sending hk
324     SET_COMMAND_CYCLE(DO_COMMAND_SENDHK);
325     #endif /* MATLAB_MEX_FILE */
326   }
327 #endif /*  MDL_START */
328
329 #ifndef MATLAB_MEX_FILE
330 // Get one packet from buffer if it is complete: Search buffer and write packet without
331 // startbyte (STX) and end byte (EOX). Start copying to packet from position initial
      position.
332 // This also indicates that the start byte STX is found already.
333 int getReplyPacket(char *buffer,int len_buf,char *packet, int len_pack, int
      initial_position)
334 {
335     int i=0,found_STX=0,found_EOX=0,packet_position=0;//found_STX_command=0;
336     // Go through complete buffer
337     for (i=0;i< len_buf;i++)
338     {
339         // If indication is set for finding the STX or if it is already in packet from
              last run.
340         if (found_STX || initial_position > 0)
341         {
342             // When I'm here, I already found STX.
343             if (buffer[i]==EOX)
344             {
345                 // Now a complete packet is received. But don't write in
346                 // packet (packet is without STX/EOX).
347                 // Set indication for a found end byte.
348                 found_EOX=1;
349                 // found end of packet, return length of received packet
350                 return initial_position + packet_position;
351             }
352             else
353             {
354                 if (initial_position + packet_position <= len_pack)
355                 {
356                     // If no EOX is found, copy buffer to packet when packet array is
                          big enough.
357                     packet[initial_position + packet_position++]=buffer[i];
358                 }
359                 else
360                 {
361                     //packet is too small, this should not happen.
362                     printf("Error: packet array too small to handle received packet\n")
                          ;
363                 }
364             }
365         }
366         else
367         {
368             // STX not found until now.
369             if (buffer[i]==STX_reply)
370             {
371                 // Now STX found, and start copying buffer to packet array.
372                 found_STX=1;
373             }
```

```
374             }
375         }
376         // If buffer does not include EOX or STX_reply
377         if (!found_EOX && !found_STX)
378         {
379             //Can happen if reply is too slow, no problem and if echo is received but no
                    answere.
380             #ifdef DEBUG
381             printf("Error: neither STX nor EOX, or just echo received!\n");
382             #endif
383             return 0;
384         }
385         // Ff buffer includes a packet start byte but no packet end byte (part of packet)
386         if (found_STX && !found_EOX)
387         {
388             // Return length of packet part (negative)
389             return (initial_position + packet_position)*-1;
390         }
391     }
392
393     int set_checksum(char *packet, int len_pack)
394     {
395         // A very sensless definition of a checksum:
396         // Add all wanted bytes together, take the last two bytes of the
397         // resulting integer, split this int intow two bytes (upper and lower)
398         // convert them from hex notation into the corresponding ascii char:
399         // 0x1 -> '1'
400         // 0x9 -> '9'
401         // 0xa -> 'A' (0x65)
402         // ...
403         // these two chars are the 'checksum'
404         int i,helpsum=0;
405         // Go through packet (without the checksum bytes)
406         for (i=0;i<len_pack-3;i++)
407         {
408             // Add them together
409             helpsum+=(0x000000FF & packet[i]);
410         }
411         // Use just the lower to bytes of the sum
412         helpsum=(0x0000FFFF & helpsum);
413         int a,b;
414         // Take the upper byte and calculate the ASCII value, e.g. 3 -> 48 +3 = '3'
415         a=hex2nibble((0x000000F0 & helpsum)>>4);
416         // The same for the lower byte
417         b=hex2nibble((0x0000000F & helpsum));
418         // Set the checksum in the packet
419         packet[len_pack-3]=a;
420         packet[len_pack-2]=b;
421         return 0;
422     }
423     int check_checksum(char *packet, int len_pack,int stx)
424     {
425         // The same as set_checksum, but check if the set checksum is right.
426
427         int i,helpsum=stx;
428         for (i=0;i<len_pack-2;i++)
429         {
430             helpsum+=(0x000000FF & packet[i]);
431         }
432         helpsum=(0x0000FFFF & helpsum);
433         int a,b;
434
```

```
435     a=hex2nibble((0x000000F0 & helpsum)>>4);
436     b=hex2nibble((0x0000000F & helpsum));
437     if (a=packet[len_pack-2] && b==packet[len_pack-1])
438     {
439         // Checksum ok.
440         return 1;
441     }
442     else
443     {
444         // Checksum wrong!
445         return 0;
446     }
447 }
448
449 int hex2nibble(int hex)
450 {
451     // Calculate the corresponding ASCII value
452     // from a nibbel (4bits: 0,..,9,a,..,f)
453     // Uppercase for A..F
454     if (hex < 10)
455     {
456         return hex+48;
457     }
458     else
459     {
460         return hex+65-10;
461     }
462 }
463 int nibble2hex(int nibble)
464 {
465     // Calculate the nibbel from a char (ASCII)
466     if (65 <= nibble && nibble <= 70)
467     {
468         return nibble-65+10;
469     }
470     else
471     {
472         return nibble-48;
473     }
474 }
475
476 uint16_T getValue_uint16(char * packet)
477 {
478     // Calculate value of a received packet by decoding the ASCII to nibble notation
479     // uint16 means unsigned integer with 16bits. This means in ascii notation 32 bit
        are to be reduced to 16
480     return nibble2hex(packet[0]) << 12  | nibble2hex(packet[1]) << 8 | nibble2hex(
        packet[2]) << 4  | nibble2hex(packet[3]);
481 }
482 int16_T getValue_int16(char * packet)
483 {
484     // same as uint16 but with a signed interger.
485     return nibble2hex(packet[0]) << 12  | nibble2hex(packet[1]) << 8 | nibble2hex(
        packet[2]) << 4  | nibble2hex(packet[3]);
486 }
487
488 int getValue_int32(char * packet)
489 {
490     // same as int16 but with a 32bit integer. 8 ascii chars are needed.
491     return nibble2hex(packet[0]) << 28  | nibble2hex(packet[1]) << 24 | nibble2hex(
        packet[2]) << 20  | nibble2hex(packet[3]) << 16 |nibble2hex(packet[4]) << 12
        | nibble2hex(packet[5]) << 8 | nibble2hex(packet[6]) << 4  | nibble2hex(packet
```

```
        [7]);
}
int setencValue_int32(char * packet, int position, int value)
{
    // encode a 32bit integer to 8 chars.
    packet[position +0]=hex2nibble((0xF0000000 & value)>>28);
    packet[position +1]=hex2nibble((0xF000000 & value)>>24);
    packet[position +2]=hex2nibble((0xF00000 & value)>>20);
    packet[position +3]=hex2nibble((0xF0000 & value)>>16);
    packet[position +4]=hex2nibble((0xF000 & value)>>12);
    packet[position +5]=hex2nibble((0xF00 & value)>>8);
    packet[position +6]=hex2nibble((0xF0 & value)>>4);
    packet[position +7]=hex2nibble((0xF & value));
    return 0;
}

int getRWnumber(SimStruct *S,int hex)
{
    // The reaction wheels are cycled. The are cycled by values 0,1,2,..n;
    // This function returns the cycle number by theire Adress, e.g. 0x7b.
    // Identifying the addresser of a packet.
    int i=0,rw_hex;
    for (i=0;i<NUM_RW;i++)
    {
        rw_hex =(uint32_T) mxGetScalar(RWADDRESSES + i);
        if (rw_hex==hex)
        {
            return i;
        }
    }
    return -1;
}
#endif
/* Function: mdlOutputs =======================================================
 * Abstract:
 *    In this function, you compute the outputs of your S-function
 *    block.
 */
static void mdlOutputs(SimStruct *S, int_T tid)
{
    #ifndef MATLAB_MEX_FILE
    //int packbufsize=PACKETBUFFERSIZE;
    //char packet[packbufsize];
    //bzero(packet,packbufsize);
    char * packet = RXBUFFER;
    //-----------------------------------------------------------------------//
    // Input port signals
    const double    *control = (const double *) ssGetInputPortSignal(S,0);
    const uint8_T   *controlmode = (const uint8_T *) ssGetInputPortSignal(S,1);
    const uint8_T   *in_power_index = (const uint8_T *) ssGetInputPortSignal(S,2);
    const bool   *clearcontroldeviation = (const bool *) ssGetInputPortSignal(S,3);
    const uint8_T   *RXBytes     = (const uint8_T *) ssGetInputPortSignal(S,4);
    const uint32_T  *getNumRXBytes     = (const uint32_T *) ssGetInputPortSignal(S,5);
    const uint32_T  *getNumTXBytes     = (const uint32_T *) ssGetInputPortSignal(S,7);
    const bool   *restart_controller = (const bool *) ssGetInputPortSignal(S,8);
    //-----------------------------------------------------------------------//
    //-----------------------------------------------------------------------//
    // Output port signals
    uint8_T *TXBytes = (uint8_T *) ssGetOutputPortRealSignal(S,13);
    uint32_T *setNumTXBytes = (uint32_T *) ssGetOutputPortRealSignal(S,14);
    uint32_T *setNumRXBytes = (uint32_T *) ssGetOutputPortRealSignal(S,15);

```

```
553     // Telemetry and houskeeping
554     double *omegaDemanded = (double *) ssGetOutputPortRealSignal(S,0);
555     double *deTheta = (double *) ssGetOutputPortRealSignal(S,1);
556     double *limlow = (double *) ssGetOutputPortRealSignal(S,2);
557     double *limhigh = (double *) ssGetOutputPortRealSignal(S,3);
558     double *omegaEstimated = (double *) ssGetOutputPortRealSignal(S,4);
559     //Housekeeping
560     double *voltage = (double *) ssGetOutputPortRealSignal(S,5);
561     double *current = (double *) ssGetOutputPortRealSignal(S,6);
562     double *temp_motor = (double *) ssGetOutputPortRealSignal(S,7);
563     double *temp_electronic = (double *) ssGetOutputPortRealSignal(S,8);
564     uint16_T *status_bits = (uint16_T *) ssGetOutputPortRealSignal(S,9);
565     uint8_T *power_index = (uint8_T *) ssGetOutputPortRealSignal(S,10);
566     double *friction = (double *) ssGetOutputPortRealSignal(S,11);
567     double *omega = (double *) ssGetOutputPortRealSignal(S,12);
568
569     double *deltaomega = (double *) ssGetOutputPortRealSignal(S,16);
570     // Get currently-in-use RW address by the CYCLE variable.
571     int current_rw = (uint32_T) mxGetScalar(RWADDRESSES + RW_CYCLE);
572
573
574     // Increment ticktocks to calculate time between two commands to
575     // the same rw.
576     int i=0;
577     int * ticktock = (int *) TICKTOCK;
578     for(i=0;i<NUM_RW;i++)
579     {
580         ticktock[i]++;
581     }
582
583     // Special requests from outside restart/(powermode)/clear control deviation
584     // Clear the deviation.
585     if (clearcontroldeviation[0] && COMMAND_CYCLE != DO_COMMAND_SENDHK)
586     {
587         SET_COMMAND_CYCLE(DO_COMMAND_CLEAR_DELTA_TH);
588     }
589     if (restart_controller[0] && COMMAND_CYCLE != DO_COMMAND_SENDHK)
590     {
591         SET_COMMAND_CYCLE(DO_COMMAND_RESTART);
592     }
593
594     // Funzt nicht kommen komische werte vom RW zurueck 0=0,7 1=23 2=39...
595     //if (in_power_index[RW_CYCLE] != power_index[RW_CYCLE] && COMMAND_CYCLE !=
            DO_COMMAND_SENDHK) // first HK has to be updated
596     //{
597     //    SET_COMMAND_CYCLE(DO_COMMAND_SET_POWER);
598     //}
599
600     // Sending Modus 1:
601     if (MODUS == 0)
602     {
603         // Cycle through all COMMAND: First ask for houskeepin, second for "DATA" and
                then command new rpm or rpm/s.
604         // Or the special cases are requested
605         switch (COMMAND_CYCLE)
606         {
607             case DO_COMMAND_SENDHK:
608             {
609                 // Build requesting packet for houskeeping data of the reaction wheel
                        currently in cycle (different than command cycle)
610                 char packet_str[]={STX_command,current_rw,ADR_master,COM_SENDHK,0x00,0
                        x00,0xd};
```

```
611                 // Set checksum in packet string.
612                 set_checksum(packet_str,7);
613                 int i=0;
614                 // Copy packet to output vector.
615                 for (i=0;i<7;i++)
616                 {
617                     TXBytes[i]=packet_str[i];
618                 }
619                 // Set number of bytes to transmit.
620                 setNumTXBytes[0]=7;
621                 // Get everything in return. If less its ok too.
622                 setNumRXBytes[0]=MAXRXBYTES;
623                 // Change to receiving mode.
624                 SET_MODUS(1);
625                 // Next packet to be send (after the reply to this packet is received)
                        is DATA.
626                 SET_COMMAND_CYCLE(DO_COMMAND_SENDDATA);
627                 break;
628             }
629             case DO_COMMAND_SENDDATA:
630             {
631                 // similiar to houskeeping (DO_COMMAND_SENDHK).
632                 char packet_str[]={STX_command,current_rw,ADR_master,COM_SENDDATA,0x00
                        ,0x00,0xd};
633                 set_checksum(packet_str,7);
634                 int i=0;
635                 for (i=0;i<7;i++)
636                 {
637                     TXBytes[i]=packet_str[i];
638                 }
639                 setNumTXBytes[0]=7;
640                 setNumRXBytes[0]=MAXRXBYTES;
641                 SET_MODUS(1);
642                 SET_COMMAND_CYCLE(DO_COMMAND_CONTROL_VALUE);
643                 break;
644             }
645             case DO_COMMAND_CONTROL_VALUE:
646             {
647                 // Send a new control value to the reaction wheel in cycle.
648                 // Do this just if time is bigger than 0.5 seconds (
                        TIME_BETWEEN_CONTROL_COMMANDS)
649                 if (ssGetSampleTime(S,0)*ticktock[RW_CYCLE] >
                        TIME_BETWEEN_CONTROL_COMMANDS)
650                 {
651                     #ifdef DEBUG
652                     //write delta time between commanding rw
653                     printf("DEBUG: RW=%d(0x%x), Time between commands: %f\n",RW_CYCLE,
                            current_rw,ssGetSampleTime(S,0)*ticktock[RW_CYCLE]);
654                     #endif
655                     // ticktock to zero
656                     ticktock[RW_CYCLE]=0;
657                     char control_mode=0;
658                     double unitdivisor=1;
659                     // Theire are different modes on which the reaction wheels can be
                            utilized.
660                     switch (controlmode[RW_CYCLE])
661                     {
662                         case MODE_SET_OMEGA_SIMPLE:
663                         {
664                             // simple mode with given angular rate
665                             control_mode =COM_SET_OMEGA_SIMPLE;
666                             // The transmitted values are integers which are scaled by
```

```
                        a predefined value (unitdevisor)
667                     unitdivisor=0.12;
668                     break;
669                 }
670                 case MODE_SET_OMEGA_STRAT:
671                 {
672                     // strat mode with given angular rate
673                     control_mode =COM_SET_OMEGA_STRAT;
674                     unitdivisor=0.12;
675                     break;
676                 }
677                 case MODE_SET_DEOMEGA_COARSE:
678                 {
679                     // coarse mode with change in angular rate
680                     control_mode =COM_SET_DEOMEGA_COARSE;
681                     unitdivisor=0.000125*60; //rpm/s
682                     break;
683                 }
684                 case MODE_SET_DEOMEGA_ADAPT:
685                 {
686                     // adaptive mode with change in angular rate
687                     control_mode =COM_SET_DEOMEGA_ADAPT;
688                     unitdivisor=0.000125*60; //rpm/s
689                     break;
690                 }
691             }
692             // Build packet for control command
693             char packet_str[]={STX_command,current_rw,ADR_master,control_mode,0
                    x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0xd};
694             // Calculate integer value of desired command value.
695             // integer division, _loss of decimal places_;
696             int control_int=(int) (control[RW_CYCLE]/unitdivisor);
697             // Calculate ASCII from integer
698             setencValue_int32(packet_str, 4 ,control_int);
699             // Set the checksum.
700             set_checksum(packet_str,15);
701             int i=0;
702             // Copy packet on output.
703             for (i=0;i<15;i++)
704             {
705                 TXBytes[i]=packet_str[i];
706             }
707             // Set number of bytes to be transmitted
708             setNumTXBytes[0]=15;
709             // Number of bytes to receive.
710             setNumRXBytes[0]=MAXRXBYTES;
711             // Next modus is receiving.
712             SET_MODUS(1);
713         }
714         // Cycle to next reaction wheel, if this is the last one, goto the
                zeroth
715         if (RW_CYCLE >= NUM_RW-1)
716         {
717             SET_RW_CYCLE(0);
718         }
719         else
720         {
721             SET_RW_CYCLE(RW_CYCLE+1);
722         }
723         // Next reaction wheel is set no start over with requesting houskeeping
                ...
724         SET_COMMAND_CYCLE(DO_COMMAND_SENDHK);
```

```
725                 break;
726             }
727             case DO_COMMAND_CLEAR_DELTA_TH:
728             {
729                 // Special: Build packet to notify the reaction wheel to clear delta
                        theta.
730                 char packet_str[]={STX_command,current_rw,ADR_master,COM_CLEAR_DELTA_TH
                        ,0x00,0x00,0xd};
731                 set_checksum(packet_str,7);
732                 int i=0;
733                 for (i=0;i<7;i++)
734                 {
735                     TXBytes[i]=packet_str[i];
736                 }
737                 setNumTXBytes[0]=7;
738                 setNumRXBytes[0]=MAXRXBYTES;
739                 SET_MODUS(1);
740                 SET_COMMAND_CYCLE(DO_COMMAND_SENDHK);
741                 break;
742             }
743             case DO_COMMAND_SET_POWER:
744             {
745                 // Special: Switch powermode
746                 char packet_str[]={STX_command,current_rw,ADR_master,COM_SET_POWER,0x00
                        ,0x00,0x00,0xd};
747                 packet_str[4]=hex2nibble(in_power_index[RW_CYCLE]);
748                 printf("bla:␣%c,␣%d␣\n",packet_str[4],in_power_index[RW_CYCLE]);
749                 set_checksum(packet_str,8);
750                 int i=0;
751                 for (i=0;i<8;i++)
752                 {
753                     TXBytes[i]=packet_str[i];
754                 }
755                 setNumTXBytes[0]=8;
756                 setNumRXBytes[0]=MAXRXBYTES;
757                 SET_MODUS(1);
758                 SET_COMMAND_CYCLE(DO_COMMAND_SENDHK);
759                 break;
760             }
761             case DO_COMMAND_RESTART:
762             {
763                 // Special: Restart microcontroller in reaction wheel.
764                 char packet_str[]={STX_command,current_rw,ADR_master,COM_RESTART,'A','D
                        ','B','C',0x00,0x00,0xd};
765                 set_checksum(packet_str,11);
766                 int i=0;
767                 for (i=0;i<11;i++)
768                 {
769                     TXBytes[i]=packet_str[i];
770                 }
771                 setNumTXBytes[0]=11;
772                 setNumRXBytes[0]=MAXRXBYTES;
773                 SET_MODUS(1);
774                 SET_COMMAND_CYCLE(DO_COMMAND_SENDHK);
775                 break;
776             }
777         }
778     }
779     // Receive Modus
780     if (MODUS == 1)
781     {
782         // Increase watchdog: If waiting for a packet but in a defined period of time (
```

```
        MAX_REPLY_WAIT)
783     // nothing comes back... go back to sending modus 1. A restart does not send a
            packet (its a guess).
784     SET_PACKET_REPLY_WATCHDOG(PACKET_REPLY_WATCHDOG+1);
785     if (PACKET_REPLY_WATCHDOG * ssGetSampleTime(S,0) > MAX_REPLY_WAIT)
786     {
787         SET_MODUS(0);
788         SET_PACKET_REPLY_WATCHDOG(0);
789         printf("Error: Requested packet didn't show up![or microcontroller restart
                ]\n");
790     }
791     // If the transmitted number of bytes is equal to the wanted number -> stop
            sending them.
792     // If two runs are needed to send the packet this will not work!.
793     // But this should not happen.
794     if (getNumTXBytes[0]==setNumTXBytes[0])
795     {
796         setNumTXBytes[0]=0;
797     }
798     else
799     {
800         #ifdef DEBUG
801             if (getNumTXBytes[0] >0)
802             {
803                 printf("Error: Requested number of bytes to send are not send (not
                        considered in program... not good.).\n");
804             }
805         #endif
806     }
807     int i=0,status_getReplyPacket=0;
808     // If bytes are received
809     if (getNumRXBytes[0] >0)
810     {
811         // Try getting information and write them in the packet buffer.
812         status_getReplyPacket=getReplyPacket((char*) RXBytes,(int) getNumRXBytes
                [0],packet,MAXRXBYTES, RXBUFFER_POSITION);
813         // If the status of the trying is negetiv, then only a parte of the packet
                is received.
814         if (status_getReplyPacket < 0)
815         {
816             status_getReplyPacket*=-1;
817             // Save position in packet buffer for data received in next run.
818             SET_RXBUFFER_POSITION(status_getReplyPacket);
819         }
820         else
821         {
822             // Positive status: Complete packet in buffer.
823             // Check the checksum if whole packet bigger than the checksum size of
                    two bytes.
824             int chk=check_checksum(packet,status_getReplyPacket,STX_reply);
825             if(!chk && status_getReplyPacket > 0) //Checksum allright and packet
                    longer than zero bytes?
826             {
827                 printf("Error: Checksum error in packet from reaction wheel[
                        status_getReplyPacket=%d,RXBUFFER_POSITION=%d].\n",
                        status_getReplyPacket,RXBUFFER_POSITION);
828                 for (i=0;i<30;i++)
829                  {
830                      printf("0x%x,",(0x000000FF & packet[i]));
831                      if (i==29)
832                      {
833                          printf("\n bytes in buffer=%d\n",getNumRXBytes[0]);
```

```
834                         }
835
836                     }
837                     for (i=0;i<getNumRXBytes[0];i++)
838                      {
839                          printf("0x%x,",(0x000000FF & RXBytes[i]));
840                          if (i==getNumRXBytes[0]-1)
841                          {
842                              printf("\n");
843                          }
844
845                     }
846                     // Forget _this corrupt_ packet go on with your life.
847                     // Set packet_position back to zero.
848                     SET_RXBUFFER_POSITION(0);
849                     // Go back to transmision modus.
850                     SET_MODUS(0);
851                     // Zero out old packet.
852                     bzero(packet,MAXRXBYTES);
853                     // Reset watchdog.
854                     SET_PACKET_REPLY_WATCHDOG(0);
855                 }
856
857                 if (chk==1)
858                 {
859                     // Checksum allright.
860                     // Now decrypt the packet. The third byte is the descriptor of the
                            packet (with a bit flip on the highes bit)
861                     switch (0x000000FF & (packet[2] ^ 0b10000000))
862                     {
863                         case COM_SENDHK:
864                         {
865                             // The packet includes houskeeping data.
866                             // Get rw number from addreeser from packet header.
867                             int rw=getRWnumber(S,packet[1]);
868                             #ifdef DEBUG
869                             if(rw <0)
870                             {
871                                 printf("Error:␣Could␣not␣determine␣the␣reaction␣wheel␣
                                        number␣from␣packet␣header.\n");
872                             }
873                             #endif
874                             // Calculate outputs from received data
875                             //printf("Houskeeping:\n");
876                             //printf("Zwischenkreisspannung: %fV\n",0.01*
                                    getValue_uint16(packet+3));
877                             voltage[rw]=0.01*getValue_uint16(packet+3);
878                             //printf("Zwischenkreisstrom: %fA\n",0.01*getValue_uint16(
                                    packet+7));
879                             current[rw]=0.01*getValue_uint16(packet+7);
880                             //printf("Temp Motor: %f deg\n",0.1*getValue_int16(packet
                                    +11));
881                             temp_motor[rw]=0.1*getValue_int16(packet+11);
882                             //printf("Temp elektronik: %f deg\n",0.1*getValue_int16(
                                    packet+15));
883                             temp_electronic[rw]=0.1*getValue_int16(packet+15);
884                             status_bits[rw]=0.1*getValue_uint16(packet+19);
885                             // decode status bits:
886                             if ((status_bits[0] & (1<<15)))
887                             {
888                              printf("Error:␣Reaction␣wheel␣0x%x␣EE_ROM␣loading␣Error.\n
                                     ");
```

```
889                     }
890                     if ((status_bits[0] & (1<<14)))
891                     {
892                      printf("Warning: Reaction wheel 0x%x temperature warning.\
                            n");
893                     }
894                     if ((status_bits[0] & (1<<13)))
895                     {
896                      printf("Error: Reaction wheel 0x%x temperatur error. Set
                            power mode 0.\n");
897                     }
898                     //printf("Power index: %d[]\n",getValue_uint16(packet+23));
899                     power_index[rw]=getValue_uint16(packet+23);
900                     //printf("Z1: %d[]\n",getValue_int16(packet+27));
901                     friction[rw]=getValue_uint16(packet+27);
902                     //printf("OmegaMeasured: %f[rpm]\n",.12*getValue_int32(
                            packet+31));
903                     omega[rw]=0.12*getValue_int32(packet+31);
904                     // Calculate derivative of the angulare rate only when the
                            angular rate change in the last cycle
905                     // The values are not updated very often so sometimes the
                            same value is send twice.
906                     double * lastomega=LAST_OMEGA;
907                     if ((lastomega[rw]-omega[rw])*(lastomega[rw]-omega[rw]) <
                            0.0001 || START_DERIVATIVES )
908                     {
909                         START_DERIVATIVES=0;
910                     }
911                     else
912                     {
913                         // Value has change.
914                         struct timeval * deltaclock=DELTA_CLOCK;
915                         struct timeval now;
916                         // Get time of day.
917                         gettimeofday(&now,0);
918                         // Calculate time difference to last time the angular
                                rate was requested (for this particular reaction
                                wheel)
919                         double dt=((double) (now.tv_sec-deltaclock[rw].tv_sec))
                                +1./1000000*((double) (now.tv_usec-deltaclock[rw].
                                tv_usec));
920                         // Set output
921                         deltaomega[rw]=(omega[rw]-lastomega[rw])/dt;
922                         // numerical derivations are evil, so try if it is a
                                number if not its zero.
923                         if (isnan(deltaomega[rw])) deltaomega[rw]=0.;
924                         if (isinf(deltaomega[rw])) deltaomega[rw]=0.;
925                         // Save the last angular rate.
926                         lastomega[rw]=omega[rw];
927                         // Save the time.
928                         gettimeofday(&deltaclock[RW_CYCLE],0);
929                     }
930                     break;
931                 }
932                 case COM_SENDDATA:
933                 {
934                     // Received packti are DATA.
935                     // Get rw number from addreeser from packet header.
936                     int rw=getRWnumber(S,packet[1]);
937                     #ifdef DEBUG
938                     if(rw <0)
939                     {
```

```
940                         printf("Error: Could not determine the reaction wheel
                                number from packet header.\n");
941                     }
942                     #endif
943                     // Set corresponding outputs.
944                     omegaDemanded[rw]=0.12*getValue_int32(packet+3);
945                     deTheta[rw]=360./2000*getValue_int32(packet+11);
946                     limlow[rw]=60*0.008*getValue_int16(packet+19);
947                     limhigh[rw]=60*0.008*getValue_int16(packet+23);
948                     omegaEstimated[rw]=0.12*getValue_int32(packet+27);
949                     break;
950                 }
951                 case 0x7a: //0xfa 'encrypted'
952                 {
953                     // An error packet.
954                     printf("Error: Maybe checksum error in packet _to_ reaction
                            wheel.\n");
955                     break;
956                 }
957                 case 0x78: //0xf8 'encrypted'
958                 {
959                     // An error packet
960                     printf("Error: Still working on last command.\n");
961                     break;
962                 }
963                 case COM_SET_OMEGA_SIMPLE: // omegasimple reply
964                 {
965                     // Just a reply for a commanded value. Do nothing.
966                     break;
967                 }
968                 case COM_SET_OMEGA_STRAT:
969                 {
970                     // Just a reply for a commanded value. Do nothing.
971                     break;
972                 }
973                 case COM_SET_DEOMEGA_COARSE:
974                 {
975                     // Just a reply for a commanded value. Do nothing.
976                     break;
977                 }
978                 case COM_SET_DEOMEGA_ADAPT:
979                 {
980                     // Just a reply for a commanded value. Do nothing.
981                     break;
982                 }
983                 case COM_CLEAR_DELTA_TH:
984                 {
985                     // Reply packet for clearing delta theta
986                     // Get rw number from addreeser from packet header.
987                     int rw=getRWnumber(S,packet[1]);
988                     printf("Cleared DELTA_TH for reaction wheel: %d (actually
                            not sure if this is working.)\n",rw);
989                     break;
990                 }
991                 case COM_SET_POWER:
992                 {
993                     // Reply packet for power index change
994                     // Get rw number from addreeser from packet header.
995                     int rw=getRWnumber(S,packet[1]);
996                     printf("Set power index for reaction wheel: %d\n",rw);
997                     break;
998                 }
```

```
999                         default:
1000                        {
1001                            // Command not known.
1002                            printf("Warning: Reply not recognized (0x%x)!\n",0x000000FF
                                    & (packet[2] ^ 0b10000000));
1003                        }
1004
1005                    }
1006                    // Set packet_position back to zero.
1007                    SET_RXBUFFER_POSITION(0);
1008                    // Go back to transmision modus.
1009                    SET_MODUS(0);
1010                    // Zero out old packet.
1011                    bzero(packet,MAXRXBYTES);
1012                    // Reset watchdog.
1013                    SET_PACKET_REPLY_WATCHDOG(0);
1014                }
1015            }
1016        }
1017    }
1018
1019    #endif /* MATLAB_MEX_FILE */
1020 }
1021
1022
1023
1024 #undef MDL_UPDATE  /* Change to #undef to remove function */
1025 #if defined(MDL_UPDATE)
1026   /* Function: mdlUpdate ======================================================
1027    * Abstract:
1028    *    This function is called once for every major integration time step.
1029    *    Discrete states are typically updated here, but this function is useful
1030    *    for performing any tasks that should only take place once per
1031    *    integration step.
1032    */
1033   static void mdlUpdate(SimStruct *S, int_T tid)
1034   {
1035   }
1036 #endif /* MDL_UPDATE */
1037
1038
1039
1040 #undef MDL_DERIVATIVES  /* Change to #undef to remove function */
1041 #if defined(MDL_DERIVATIVES)
1042   /* Function: mdlDerivatives =================================================
1043    * Abstract:
1044    *    In this function, you compute the S-function block's derivatives.
1045    *    The derivatives are placed in the derivative vector, ssGetdX(S).
1046    */
1047   static void mdlDerivatives(SimStruct *S)
1048   {
1049   }
1050 #endif /* MDL_DERIVATIVES */
1051
1052
1053
1054 /* Function: mdlTerminate =====================================================
1055  * Abstract:
1056  *    In this function, you should perform any actions that are necessary
1057  *    at the termination of a simulation.  For example, if memory was
1058  *    allocated in mdlStart, this is the place to free it.
1059  */
```

```
1060  static void mdlTerminate(SimStruct *S)
1061  {
1062      #ifndef MATLAB_MEX_FILE
1063    // Free stuff
1064    free(TICKTOCK);
1065    free(DELTA_CLOCK);
1066    free(LAST_OMEGA);
1067    free(RXBUFFER);
1068      #endif /* MATLAB_MEX_FILE */
1069  }
1070
1071
1072  /*====================================================*
1073   * See sfuntmpl_doc.c for the optional S-function methods *
1074   *====================================================*/
1075
1076  /*============================*
1077   * Required S-function trailer *
1078   *============================*/
1079
1080  #ifdef  MATLAB_MEX_FILE    /* Is this file being compiled as a MEX-file? */
1081  #include "simulink.c"      /* MEX-file interface mechanism */
1082  #else
1083  #include "cg_sfun.h"       /* Code generation registration function */
1084  #endif
```

C.2 LITEF µFORS-6U fiber optical gyroscope

C.2.1 Simulink™ block

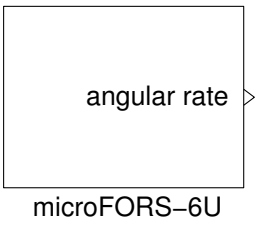


Figure C.2: Simulink™ block for the µFORS-6U software interface

C.2.2 Source code

Listing C.2: HWI_microFORS_6U_sf.c

```
1  // HINT in simulink num rxbytes +1, damit erkannt werden kann wenn noch mehr im buffer
       lagern.
2
3  #define S_FUNCTION_NAME  HWI_microFORS_6U_sf
4  #define S_FUNCTION_LEVEL 2
5
6  /*
7   * Need to include simstruc.h for the definition of the SimStruct and
8   * its associated macro definitions.
9   */
10 #include "simstruc.h"
11
12 // DEBUG
13 //#define DEBUG 1
14 //------
15
16 #ifndef MATLAB_MEX_FILE
17 // include files here
18 #endif
19
20 static void mdlInitializeSizes(SimStruct *S)
21 {
22     // Parameters from the mask
23     ssSetNumSFcnParams(S, 4);  /* Number of expected parameters: thenumofbytes,themodus
           ,thereceivingdataandrange,theidentifier */
24     if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) {
25         /* Return if number of expected != number of actual parameters */
26         return;
27     }
28     // 3 input ports:
29     if (!ssSetNumInputPorts(S, 3)) return;
30     /*Input Port 0: a vector of chars from the serial line */
31     ssSetInputPortWidth(S, 0,(int) mxGetScalar(ssGetSFcnParam(S,0)));
32     ssSetInputPortDataType(S, 0, SS_UINT8);
33     ssSetInputPortComplexSignal(S,  0, COMPLEX_NO);
34     ssSetInputPortRequiredContiguous(S, 0, true); /*direct input signal access*/
35     ssSetInputPortDirectFeedThrough(S, 0, 1);
```

```
36      /*Input Port 1: the number of chars received on the serial line */
37      ssSetInputPortWidth(S, 1, 1);
38      ssSetInputPortDataType(S, 1, SS_UINT32);
39      ssSetInputPortComplexSignal(S,  1, COMPLEX_NO);
40      ssSetInputPortRequiredContiguous(S, 1, true); /*direct input signal access*/
41      ssSetInputPortDirectFeedThrough(S, 1, 1);
42      /*Input Port 2: status port from serial RX */
43      ssSetInputPortWidth(S, 2, 1);
44      ssSetInputPortDataType(S, 2, SS_INT32);
45      ssSetInputPortComplexSignal(S,  2, COMPLEX_NO);
46      ssSetInputPortRequiredContiguous(S, 2, true); /*direct input signal access*/
47      ssSetInputPortDirectFeedThrough(S, 2, 1);
48      // 1 output port:
49      if (!ssSetNumOutputPorts(S, 1)) return;
50      /* Output Port 0: double value describing a rate or angle  */
51      ssSetOutputPortWidth(S, 0, 1);
52      ssSetOutputPortDataType(S, 0, SS_DOUBLE);
53      ssSetOutputPortComplexSignal(S, 0, COMPLEX_NO);
54      // global variables:
55      ssSetNumRWork(S, 2); //real (double): range,LSB
56      ssSetNumIWork(S, 6); //integer: resolution,modus,NUM_BYTES_OF_PACKET,CHKSUMERRORS,
            BUFFER_POSITION
57      ssSetNumPWork(S, 1); //packet_buffer
58  }
59
60  /* Function: mdlInitializeSampleTimes =========================================
61   * Abstract:
62   *    This function is used to specify the sample time(s) for your
63   *    S-function. You must register the same number of sample times as
64   *    specified in ssSetNumSampleTimes.
65   */
66  static void mdlInitializeSampleTimes(SimStruct *S)
67  {
68      ssSetSampleTime(S, 0, INHERITED_SAMPLE_TIME);
69      ssSetOffsetTime(S, 0, FIXED_IN_MINOR_STEP_OFFSET);
70
71  }
72
73
74
75  #define MDL_INITIALIZE_CONDITIONS   /* Change to #undef to remove function */
76  #if defined(MDL_INITIALIZE_CONDITIONS)
77    /* Function: mdlInitializeConditions ========================================
78     * Abstract:
79     *    In this function, you should initialize the continuous and discrete
80     *    states for your S-function block.  The initial states are placed
81     *    in the state vector, ssGetContStates(S) or ssGetRealDiscStates(S).
82     *    You can also perform any other initialization activities that your
83     *    S-function may require. Note, this routine will be called at the
84     *    start of simulation and if it is present in an enabled subsystem
85     *    configured to reset states, it will be call when the enabled subsystem
86     *    restarts execution to reset the states.
87     */
88    static void mdlInitializeConditions(SimStruct *S)
89    {
90    }
91  #endif /* MDL_INITIALIZE_CONDITIONS */
92
93  // Renaming of the parameters and global variables.
94  // parameters
95  #define PARA_NUMOFBYTES ssGetSFcnParam(S,0)
96  #define PARA_MODUS ssGetSFcnParam(S,1)
```

```
97  #define PARA_RANGE ssGetSFcnParam(S,2)
98  #define PARA_IDENTIFIER ssGetSFcnParam(S,3)
99
100 // global variables
101 // Modus for the rate gyro [autonomous] is currently the only supported.
102 #define SET_MODUS(a) ssSetIWorkValue(S,0,a)
103 #define MODUS ssGetIWorkValue(S,0)
104 // Number of bytes in a information packet send by the rate gyro [4,5].
105 #define SET_NUMOFBYTES(a) ssSetIWorkValue(S,1,a)
106 #define NUMOFBYTES ssGetIWorkValue(S,1)
107 // Maximal value represented by a 2 or 3 byte integer (16bit or 24bit).
108 // Maximal value divided by 2^(16-1) or 2^(24-1) gives the smalles measurable
109 // real world value. '-1' because the range means +-range.
110 #define SET_RANGE(a) ssSetRWorkValue(S,0,a)
111 #define RANGE ssGetRWorkValue(S,0)
112 // Calculated (real world) value of the least significant bit of the integer
113 // representation send by the rate gyro.
114 // LSB * received integer = real world value in degrees or degrees per second
115 #define SET_LSB(a) ssSetRWorkValue(S,1,a)
116 #define LSB ssGetRWorkValue(S,1)
117 // Pointer to the buffer which will store the received data.
118 #define SET_PACKET_BUFFER(a) ssSetPWorkValue(S,0,a)
119 #define PACKET_BUFFER ssGetPWorkValue(S,0)
120 // Save the number of bytes copied to the buffer.
121 #define SET_NUM_BYTES_OF_PACKET(a) ssSetIWorkValue(S,2,a)
122 #define NUM_BYTES_OF_PACKET ssGetIWorkValue(S,2)
123 // Number of checksum errors in a row.
124 #define SET_CHECKSUM_ERRORS(a) ssSetIWorkValue(S,3,a)
125 #define CHECKSUM_ERRORS ssGetIWorkValue(S,3)
126 // How many bytes in buffer?
127 #define SET_BUFFER_POSITION(a) ssSetIWorkValue(S,4,a)
128 #define BUFFER_POSITION ssGetIWorkValue(S,4)
129 // How many bytes in buffer?
130 #define SET_DEBUG_COUNTER(a) ssSetIWorkValue(S,5,a)
131 #define DEBUG_COUNTER ssGetIWorkValue(S,5)
132
133 #define MDL_START  /* Change to #undef to remove function */
134 #if defined(MDL_START)
135   /* Function: mdlStart =======================================================
136    * Abstract:
137    *    This function is called once at start of model execution. If you
138    *    have states that should be initialized once, this is the place
139    *    to do it.
140    */
141   static void mdlStart(SimStruct *S)
142   {
143     #ifndef MATLAB_MEX_FILE
144     // Set the global variables.
145     SET_MODUS((int) mxGetScalar(PARA_MODUS));
146     SET_NUMOFBYTES((int) mxGetScalar(PARA_NUMOFBYTES));
147     SET_RANGE((double) mxGetScalar(PARA_RANGE));
148     // Allocate the packet buffer
149     uint8_T * packet_buffer=malloc(sizeof(uint8_T)*NUMOFBYTES*2);
150     // blank it
151     memset(packet_buffer, 0, sizeof(uint8_T)*NUMOFBYTES*2);
152     // and save pointer to global variable.
153     SET_PACKET_BUFFER(packet_buffer);
154     SET_NUM_BYTES_OF_PACKET(0);
155     SET_BUFFER_POSITION(0);
156     SET_CHECKSUM_ERRORS(0);
157     // Calculate the value of the LSB whether 16bit or 24bit
158     if (NUMOFBYTES == 5){
```

```
159             SET_LSB(RANGE / 8388608);
160         }
161         else
162         {
163             SET_LSB(RANGE / 32768);
164         }
165         // Get the chosen identifier + '\0'
166         char identifier[2];
167         mxGetString(PARA_IDENTIFIER, identifier, sizeof(char)*2);
168         printf("microFORS '%c' Rate gyro: Modus= %d, Resolution= %d, Range= %f\n",
                identifier[0],MODUS,NUMOFBYTES,RANGE);
169         SET_DEBUG_COUNTER(0);
170         #endif /* MATLAB_MEX_FILE */
171       }
172     #endif /*  MDL_START */
173
174     int get_packet_from_buffer(uint8_T * buffer, int buffer_position,int packet_size,
            uint8_T * packet)
175     {
176         int i=0,ii=0;
177         // search highest possible packet in buffer, than search backwards through buffer
178         //printf("%d\n",buffer_position-packet_size);
179         for (i=buffer_position-packet_size; i >= 0; i--)
180         {
181             uint8_T chksum=0;
182             for (ii=0;ii < packet_size;ii++)
183             {
184                 //printf("%d\n",buffer[ii+i]);
185                 chksum+=(uint8_T) buffer[ii+i];
186             }
187             if (chksum == 255)
188             {
189                 // Found packet, write in array
190                 for (ii=0;ii < packet_size;ii++)
191                 {
192                     packet[ii]=buffer[ii+i];
193                 }
194                 // return start position where packet was found
195                 return i;
196                 // stopp for-loop
197                 break;
198             }
199         }
200         // Nothing found
201         return -1;
202     }
203     void shift_pack_buff(uint8_T * buffer,int buffer_size,int shift_width)
204     {
205         int i=0;
206         for (i=shift_width;i < buffer_size;i++)
207         {
208             buffer[i-shift_width]=buffer[i];
209         }
210     }
211
212     /* Function: mdlOutputs =======================================================
213      * Abstract:
214      *    In this function, you compute the outputs of your S-function
215      *    block.
216      */
217     static void mdlOutputs(SimStruct *S, int_T tid)
218     {
```

```
219     #ifndef MATLAB_MEX_FILE
220         SET_DEBUG_COUNTER(DEBUG_COUNTER+1);
221         double *output = (double *)  ssGetOutputPortRealSignal(S,0);
222         // Get the chosen identifier + '\0'
223         char identifier[2];
224         mxGetString(PARA_IDENTIFIER,identifier,sizeof(char)*2);
225         // Get input port signals.
226         // Array of chars received on serial line.
227         const uint8_T   *RXBytes  = ssGetInputPortSignal(S,0);
228         // Number of chars received on serial line.
229         const uint32_T   *NumRXBytes  = ssGetInputPortSignal(S,1);
230
231
232         // Copy received data to PACKET_BUFFER
233         uint8_T *pack_buf=PACKET_BUFFER;
234         int RXint=0,i=0,remains=0,buffer_shift=0; // If more bytes received than buffer
                space available.
235         if (*NumRXBytes > NUMOFBYTES*2-BUFFER_POSITION)
236             {
237                 // How many remaining bytes must be written to buffer?
238                 remains = *NumRXBytes-NUMOFBYTES*2-BUFFER_POSITION;
239             }
240             else
241             {
242                 remains=0;
243             }
244         for(i=0;i < *NumRXBytes-remains;i++)
245         {
246             // Copy received byte in buffer.
247             pack_buf[BUFFER_POSITION]= RXBytes[i];
248             // Increase number of bytes in packet
249             SET_BUFFER_POSITION(BUFFER_POSITION+1);
250         }
251         uint8_T packet[NUMOFBYTES];
252         bzero(packet,NUMOFBYTES);
253         int packet_found=0;
254         packet_found = get_packet_from_buffer(pack_buf, BUFFER_POSITION,NUMOFBYTES,
              packet);
255
256         if (packet_found >= 0)
257         {
258             //printf("packet: %c%c%c%c%c, %d, bufferpo=%d\n",packet[0],packet[1],packet
                  [2],packet[3],packet[4],packet_found,BUFFER_POSITION);
259             // Packet was found on starting position 'packet_found' written to packet
260             // shift buffer by packet_found+NUMOFBYTES
261             shift_pack_buff(pack_buf,NUMOFBYTES*2,packet_found+NUMOFBYTES);
262             SET_BUFFER_POSITION(BUFFER_POSITION-NUMOFBYTES);
263
264             // Calculated value from the chars in integer notation.
265             if (NUMOFBYTES == 5)
266             {
267                 // Bitwise shift and delete the most left byte of an 32bit integer.
268                 // 24bit
269                 RXint=(packet[0] << 24 & 0xFF000000)|(packet[1] << 16 & 0xFF0000)|(
                      packet[2] << 8 & 0xFF00) | 0x00 ;
270                 // The 24bit representation is shifted to the left of a 32bit
271                 // representation. A shift back is not allowed, because it is a signed
272                 // integer.
273                 RXint/=256;
274             }
275             else
276             {
```

```
277                 // Bitwise shift and delete the most left bytes of an 32bit integer.
278                 // 16bit
279                 RXint=(packet[0] << 24 & 0xFF000000)|(packet[1] << 16 & 0xFF0000)|0
                        x0000;
280                 // The 16bit representation is shifted to the left of a 32bit
281                 // representation. A shift back is not allowed, because it is a signed
282                 // integer.
283                 RXint/=65536;
284             }
285             // Set number of checksum errors in a row to 0
286             if (CHECKSUM_ERRORS > 3)
287             {
288                 printf("microFORS␣'%c'␣resynced.\n",identifier[0]);
289             }
290             SET_CHECKSUM_ERRORS(0);
291             // If the status char is different from 0... and no checksum error
292             if ((uint8_T) packet[NUMOFBYTES-2] > 0)
293             {
294                 printf("microFORS␣'%c'␣Error:␣",identifier[0]);
295                 // What error has occurred?
296                 if ((packet[NUMOFBYTES-2] & 0b00000001)>0) printf("NOGO!\n");
297                 if ((packet[NUMOFBYTES-2] & 0b00000010)>0) printf("Reset␣Acknowledge.\n
                        ");
298                 if ((packet[NUMOFBYTES-2] & 0b00000100)>0) printf("Not␣Used\n");
299                 if ((packet[NUMOFBYTES-2] & 0b00001000)>0) printf("Temprature␣Warning!\
                        n");
300                 if ((packet[NUMOFBYTES-2] & 0b00010000)>0) printf("Auxiliary␣Control␣
                        Loop␣Error!\n");
301                 if ((packet[NUMOFBYTES-2] & 0b00100000)>0) printf("Hardware␣BIT␣Error!\
                        n");
302                 if ((packet[NUMOFBYTES-2] & 0b01000000)>0) printf("Measurement␣Range␣
                        Exceeded!\n");
303                 if ((uint8_T) (packet[NUMOFBYTES-2] & 0b10000000)>0) printf("Unknown␣
                        Command!\n");
304             }
305     else
306     {
307      // In the case of any error, the sensor data will not be trusted.
308      // Write the real world rate or angle on the output port.
309      output[0]=RXint * LSB;
310     }
311
312         }
313         if (packet_found < 0)
314         {
315             // No packet found
316             // Maybe no complete packet?
317             if (BUFFER_POSITION >= NUMOFBYTES)
318             {
319                 // No packet found, but more than one packet is in buffer
320                 // must be a checksum error
321                 //output[0]=0.;
322                 printf("Warning␣microFORS␣'%c':␣Checksum␣fail[%d][#%d]!\n",identifier
                        [0],CHECKSUM_ERRORS,DEBUG_COUNTER);
323                 // forget first packet and shift pack_buff down.
324                 if (CHECKSUM_ERRORS > 3)
325                 {
326                     printf("Try␣to␣sync␣[%d,bufferposition=%d]...\n",CHECKSUM_ERRORS,
                            BUFFER_POSITION);
327                     shift_pack_buff(pack_buf,NUMOFBYTES*2,1);
328                     SET_BUFFER_POSITION(BUFFER_POSITION-1);
329                 }
```

```
330                 else
331                 {
332                     shift_pack_buff(pack_buf,NUMOFBYTES*2,NUMOFBYTES);
333                     SET_BUFFER_POSITION(BUFFER_POSITION-NUMOFBYTES);
334                 }
335                 SET_CHECKSUM_ERRORS(CHECKSUM_ERRORS+1);
336             }
337         }
338         // If buffer was full before packet was found, copy remaining bytes to buffer
339         for(i=0;i < remains;i++)
340         {
341             // Copy received byte in buffer.
342             pack_buf[BUFFER_POSITION]= RXBytes[i];
343             // Increase number of bytes in packet
344             SET_BUFFER_POSITION(BUFFER_POSITION+1);
345         }
346
347
348
349   #ifdef DEBUG
350         if ( *NumRXBytes > 0  && *NumRXBytes < NUMOFBYTES)
351         {
352             printf("Just a part of the packet received [%d],no problem.\n",*NumRXBytes)
                  ;
353         }
354   #endif
355     #endif /* MATLAB_MEX_FILE */
356 }
357
358
359
360 #undef MDL_UPDATE  /* Change to #undef to remove function */
361 #if defined(MDL_UPDATE)
362   /* Function: mdlUpdate ======================================================
363    * Abstract:
364    *    This function is called once for every major integration time step.
365    *    Discrete states are typically updated here, but this function is useful
366    *    for performing any tasks that should only take place once per
367    *    integration step.
368    */
369   static void mdlUpdate(SimStruct *S, int_T tid)
370   {
371   }
372 #endif /* MDL_UPDATE */
373
374
375
376 #undef MDL_DERIVATIVES  /* Change to #undef to remove function */
377 #if defined(MDL_DERIVATIVES)
378   /* Function: mdlDerivatives =================================================
379    * Abstract:
380    *    In this function, you compute the S-function block's derivatives.
381    *    The derivatives are placed in the derivative vector, ssGetdX(S).
382    */
383   static void mdlDerivatives(SimStruct *S)
384   {
385   }
386 #endif /* MDL_DERIVATIVES */
387
388
389
390 /* Function: mdlTerminate =====================================================
```

```
391   * Abstract:
392   *    In this function, you should perform any actions that are necessary
393   *    at the termination of a simulation.  For example, if memory was
394   *    allocated in mdlStart, this is the place to free it.
395   */
396  static void mdlTerminate(SimStruct *S)
397  {
398      #ifndef MATLAB_MEX_FILE
399    // Free stuff
400    free(PACKET_BUFFER);
401      #endif /* MATLAB_MEX_FILE */
402  }
403
404
405  /*=============================================================*
406   * See sfuntmpl_doc.c for the optional S-function methods *
407   *=============================================================*/
408
409  /*=============================*
410   * Required S-function trailer *
411   *=============================*/
412
413  #ifdef  MATLAB_MEX_FILE    /* Is this file being compiled as a MEX-file? */
414  #include "simulink.c"      /* MEX-file interface mechanism */
415  #else
416  #include "cg_sfun.h"       /* Code generation registration function */
417  #endif
```

C.3 ZARM Technik amr magnetometer

C.3.1 Simulink™ block

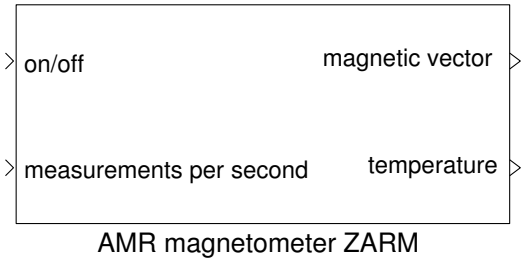


Figure C.3: Simulink™ block for the AMR magnetometer software interface

C.3.2 Source code

Listing C.3: HWI_AMR_magnetometer_sf.c

```
1  #define S_FUNCTION_NAME  HWI_AMR_magnetometer_sf
2  #define S_FUNCTION_LEVEL 2
3
4  /*
5   * Need to include simstruc.h for the definition of the SimStruct and
6   * its associated macro definitions.
7   */
8  #include "simstruc.h"
9
10 // DEBUG
11 //#define DEBUG 1
12 //-----
13
14 #ifndef MATLAB_MEX_FILE
15 // include files here
16 #endif
17
18 static void mdlInitializeSizes(SimStruct *S)
19 {
20     // Parameters from the mask
21     ssSetNumSFcnParams(S, 0);  /* Number of expected parameters: */
22     if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) {
23         /* Return if number of expected != number of actual parameters */
24         return;
25     }
26     // 3 input ports:
27     if (!ssSetNumInputPorts(S, 6)) return;
28     /*Input Port 0: a vector of chars from the serial line */
29     ssSetInputPortWidth(S, 0,9);
30     ssSetInputPortDataType(S, 0, SS_UINT8);
31     ssSetInputPortComplexSignal(S,  0, COMPLEX_NO);
32     ssSetInputPortRequiredContiguous(S, 0, true); /*direct input signal access*/
33     ssSetInputPortDirectFeedThrough(S, 0, 1);
34     /*Input Port 1: the number of chars received on the serial line */
35     ssSetInputPortWidth(S, 1, 1);
36     ssSetInputPortDataType(S, 1, SS_UINT32);
```

```
37      ssSetInputPortComplexSignal(S,  1, COMPLEX_NO);
38      ssSetInputPortRequiredContiguous(S, 1, true); /*direct input signal access*/
39      ssSetInputPortDirectFeedThrough(S, 1, 1);
40      /*Input Port 2: status port from serial RX */
41      ssSetInputPortWidth(S, 2, 1);
42      ssSetInputPortDataType(S, 2, SS_INT32);
43      ssSetInputPortComplexSignal(S,  2, COMPLEX_NO);
44      ssSetInputPortRequiredContiguous(S, 2, true); /*direct input signal access*/
45      ssSetInputPortDirectFeedThrough(S, 2, 1);
46      /*Input Port 3: switch analogue part of AMR magnetometer on/off */
47      ssSetInputPortWidth(S, 3, 1);
48      ssSetInputPortDataType(S, 3, SS_UINT8);
49      ssSetInputPortComplexSignal(S,  3, COMPLEX_NO);
50      ssSetInputPortRequiredContiguous(S, 3, true); /*direct input signal access*/
51      ssSetInputPortDirectFeedThrough(S, 3, 1);
52      /*Input Port 4: measurements per second */
53      ssSetInputPortWidth(S, 4, 1);
54      ssSetInputPortDataType(S, 4, SS_UINT8);
55      ssSetInputPortComplexSignal(S,  4, COMPLEX_NO);
56      ssSetInputPortRequiredContiguous(S, 4, true); /*direct input signal access*/
57      ssSetInputPortDirectFeedThrough(S, 4, 1);
58      /*Input Port 5: number of transmitted bytes */
59      ssSetInputPortWidth(S, 5, 1);
60      ssSetInputPortDataType(S, 5, SS_UINT32);
61      ssSetInputPortComplexSignal(S,  5, COMPLEX_NO);
62      ssSetInputPortRequiredContiguous(S, 5, true); /*direct input signal access*/
63      ssSetInputPortDirectFeedThrough(S, 5, 1);
64
65      // 5 output ports:
66      if (!ssSetNumOutputPorts(S, 5)) return;
67      // Output Port 0: magnetic vector
68      ssSetOutputPortWidth(S, 0, 3);
69      // Output Port 1: temperature
70      ssSetOutputPortWidth(S, 1, 1);
71      // Output Port 2: command to AMR
72      ssSetOutputPortWidth(S, 2, 1);
73      ssSetOutputPortDataType(S, 2, SS_UINT8);
74      ssSetOutputPortComplexSignal(S, 2, COMPLEX_NO);
75      // Output Port 3: numofbytesTX
76      ssSetOutputPortWidth(S, 3, 1);
77      ssSetOutputPortDataType(S, 3, SS_UINT32);
78      ssSetOutputPortComplexSignal(S, 3, COMPLEX_NO);
79      // Output Port 4: numofbytesRX
80      ssSetOutputPortWidth(S, 4, 1);
81      ssSetOutputPortDataType(S, 4, SS_UINT32);
82      ssSetOutputPortComplexSignal(S, 4, COMPLEX_NO);
83
84      // global variables:
85      ssSetNumIWork(S, 7); //integer: waiting, AMRstatus,number of received chars of one
            packet <9,startup,requesteddata,MODUS,assume_send
86      ssSetNumPWork(S, 1); // pointer: buffer for packet[9]
87  }
88
89  /* Function: mdlInitializeSampleTimes =========================================
90   * Abstract:
91   *    This function is used to specify the sample time(s) for your
92   *    S-function. You must register the same number of sample times as
93   *    specified in ssSetNumSampleTimes.
94   */
95  static void mdlInitializeSampleTimes(SimStruct *S)
96  {
97      ssSetSampleTime(S, 0, INHERITED_SAMPLE_TIME);
```

```
98      ssSetOffsetTime(S, 0, FIXED_IN_MINOR_STEP_OFFSET);
99
100 }
101
102
103
104 #define MDL_INITIALIZE_CONDITIONS   /* Change to #undef to remove function */
105 #if defined(MDL_INITIALIZE_CONDITIONS)
106   /* Function: mdlInitializeConditions ========================================
107    * Abstract:
108    *    In this function, you should initialize the continuous and discrete
109    *    states for your S-function block.  The initial states are placed
110    *    in the state vector, ssGetContStates(S) or ssGetRealDiscStates(S).
111    *    You can also perform any other initialization activities that your
112    *    S-function may require. Note, this routine will be called at the
113    *    start of simulation and if it is present in an enabled subsystem
114    *    configured to reset states, it will be call when the enabled subsystem
115    *    restarts execution to reset the states.
116    */
117   static void mdlInitializeConditions(SimStruct *S)
118   {
119   }
120 #endif /* MDL_INITIALIZE_CONDITIONS */
121
122 // Renaming of the parameters and global variables.
123 // parameters
124 //#define PARA_NUMOFBYTES ssGetSFcnParam(S,0)
125 //#define PARA_MODUS ssGetSFcnParam(S,1)
126 //#define PARA_RANGE ssGetSFcnParam(S,2)
127
128 // global variables
129 // Something like a watchdog. Before sending on/off command the 'watchdog'
130 // is started and one second later the the command will be send.
131 #define SET_WAITING(a) ssSetIWorkValue(S,0,a)
132 #define WAITING ssGetIWorkValue(S,0)
133 // Status of the AMR analogue part [1:=on,0:=off]
134 #define SET_AMRSTATUS(a) ssSetIWorkValue(S,1,a)
135 #define AMRSTATUS ssGetIWorkValue(S,1)
136 // Pointer to the buffer which will store the received data.
137 #define SET_PACKET_BUFFER(a) ssSetPWorkValue(S,0,a)
138 #define PACKET_BUFFER ssGetPWorkValue(S,0)
139 // Save the number of bytes copied to the buffer.
140 #define SET_NUM_BYTES_OF_PACKET(a) ssSetIWorkValue(S,2,a)
141 #define NUM_BYTES_OF_PACKET ssGetIWorkValue(S,2)
142 // For initializing the AMR magnetometer with an unkown power state
143 // a differentiation must be in place.
144 #define SET_STARTUP(a) ssSetIWorkValue(S,3,a)
145 #define STARTUP ssGetIWorkValue(S,3)
146 // Check if the requested date is equal to the last byte of the received data
147 // which is a repetition of the command. _Could_ be used for synchronisation.
148 #define SET_REQUESTED_DATA(a) ssSetIWorkValue(S,4,a)
149 #define REQUESTED_DATA ssGetIWorkValue(S,4)
150 // There are multiple modi, e.g., Receiving, Sending, Waiting ...
151 #define SET_MODUS(a) ssSetIWorkValue(S,5,a)
152 #define MODUS ssGetIWorkValue(S,5)
153 // If the status of the serial TX is ok, the 'watchdog' is started.
154 // If the answer is not received, it will assume one second later that the
155 // toggle has been completed.
156 #define SET_ASSUME_SEND(a) ssSetIWorkValue(S,6,a)
157 #define ASSUME_SEND ssGetIWorkValue(S,6)
158
159 #define MDL_START  /* Change to #undef to remove function */
```

```
160 #if defined(MDL_START)
161   /* Function: mdlStart =======================================================
162    * Abstract:
163    *    This function is called once at start of model execution. If you
164    *    have states that should be initialized once, this is the place
165    *    to do it.
166    */
167   static void mdlStart(SimStruct *S)
168   {
169     #ifndef MATLAB_MEX_FILE
170  // Initiate Buffer for maximal packet length.
171     uint32_T * packet_buffer=malloc(sizeof(uint32_T)*9);
172     memset(packet_buffer, 0, sizeof(uint32_T)*9);
173     // Make the buffer accessible globally.
174     SET_PACKET_BUFFER(packet_buffer);
175     SET_NUM_BYTES_OF_PACKET(0);
176     // Initialize global variables
177     SET_STARTUP(1);
178     SET_REQUESTED_DATA(0);
179     SET_MODUS(0);
180     SET_ASSUME_SEND(0);
181     printf("Starting AMR magnetometer.\n");
182     #endif /* MATLAB_MEX_FILE */
183   }
184 #endif /*  MDL_START */
185
186
187
188 /* Function: mdlOutputs =====================================================
189  * Abstract:
190  *    In this function, you compute the outputs of your S-function
191  *    block.
192  */
193 static void mdlOutputs(SimStruct *S, int_T tid)
194 {
195     #ifndef MATLAB_MEX_FILE
196
197     // Get all in- and output signals.
198     uint8_T *TXBytes = ssGetOutputPortSignal(S,2);
199     uint32_T *setNumTXBytes = ssGetOutputPortSignal(S,3);
200     uint32_T *setNumRXBytes = ssGetOutputPortSignal(S,4);
201     const uint8_T   *RXBytes  = ssGetInputPortSignal(S,0);
202     const uint32_T   *NumRXBytes  = ssGetInputPortSignal(S,1);
203     const uint32_T   *NumTXBytes  = ssGetInputPortSignal(S,5);
204
205     const uint8_T   *requestedamrstatus  = ssGetInputPortSignal(S,3);
206     const uint8_T   *measurementspersecond  = ssGetInputPortSignal(S,4);
207     // If not Startup, than check wether the wanted status is equal to the
208     // momentarily status of the AMR magnetometer.
209     if (STARTUP == 0)
210     {
211         if (AMRSTATUS != *requestedamrstatus && MODUS !=0 && MODUS !=1)
212         {
213             SET_MODUS(0);
214             SET_WAITING(0);
215         }
216     }
217     else
218     {
219         // IF startup: start 'watchdog'
220         TXBytes[0]=0;
221         setNumTXBytes[0]=0;
```

```
222             setNumRXBytes[0]=0;
223             SET_WAITING(WAITING+1);
224             printf("Starting up AMR magnetometer...\n");
225             if (WAITING * ssGetSampleTime(S,0) > 1)
226             {
227                 SET_WAITING(0);
228             }
229         }
230         //Modus 3: receive requested data
231         if (MODUS == 3)
232         {
233             // Did the command send properly? If so, then stop repeating yourself!
234             if (*NumTXBytes == 1)
235             {
236                 TXBytes[0]=0;
237                 setNumTXBytes[0]=0;
238             }
239             // Waiting for data (9 bytes)
240             // get packet buffer.
241             uint32_T *pack_buf=PACKET_BUFFER;
242             // Iterate through every received byte in 'RXBytes'.
243             int i=0;
244             for(i=0;i < *NumRXBytes;i++)
245             {
246                 // Save received datum to buffer.
247                 pack_buf[NUM_BYTES_OF_PACKET]=(uint32_T) RXBytes[i];
248                 // Increase bytes in Buffer value.
249                 SET_NUM_BYTES_OF_PACKET(NUM_BYTES_OF_PACKET+1);
250                 // If all 9 bytes of the packet are received ...
251                 if (NUM_BYTES_OF_PACKET == 9)
252                 {
253                     // Check if the right data is send. {Why shouoldn't??}
254                     if (pack_buf[8] == REQUESTED_DATA)
255                     {
256                         ssGetOutputPortRealSignal(S,0)[0]= (int16_T) (pack_buf[1] << 8 |
                                pack_buf[0]) * 10.0;
257                         ssGetOutputPortRealSignal(S,0)[1]= (int16_T) (pack_buf[3] << 8 |
                                pack_buf[2]) * 10.0;
258                         ssGetOutputPortRealSignal(S,0)[2]= (int16_T) (pack_buf[5] << 8 |
                                pack_buf[4]) * 10.0;
259                         ssGetOutputPortRealSignal(S,1)[0]= (int16_T) (pack_buf[7] << 8 |
                                pack_buf[6]) * 0.005519;
260                     }
261                     else
262                     {
263                         printf("Packed somehow corrupt. Either package buffer not synced or
                                communication errror on last byte (or Wrong data send == sync
                                error on whole packet), error!\n");
264                     }
265                     // A whole packet received and laid on output port, next packet can be
                            requested...
266                     SET_NUM_BYTES_OF_PACKET(0);
267                     SET_MODUS(2); // Request Modus
268                 }
269             }
270     #ifdef DEBUG
271             if ( *NumRXBytes > 0  && *NumRXBytes < 9)
272             {
273                 printf("Just a part of the packet received [%d], no problem.\n",*NumRXBytes)
                        ;
274             }
275     #endif
```

```
276     }
277
278     // Modus 0: Startup or power toggle
279     if (MODUS == 0)
280         {
281         // If startup phase switch now to normal mode.
282         SET_STARTUP(0);
283         // One second after the request for power toggle the command is
284         // beeing send.
285         if (WAITING * ssGetSampleTime(S,0) > 1){
286             // Send on or off command.
287             if (*requestedamrstatus == 1)
288             {
289                 // Send 0x99 for toggel on.
290                 TXBytes[0]=0x99;
291                 // The toggle command is one byte long.
292                 setNumTXBytes[0]=1;
293                 // The power command is acknowleged by one byte (the same as the
                        command)
294                 setNumRXBytes[0]=1;
295                 printf("switching AMR magnetometer on...\n");
296             }
297             else
298             {
299                 TXBytes[0]=0x98;
300                 setNumTXBytes[0]=1;
301                 setNumRXBytes[0]=1;
302                 // If the analogue part is beeing switched off, the output
303                 // port is set to zero.
304                 ssGetOutputPortRealSignal(S,0)[0]=0.;
305                 ssGetOutputPortRealSignal(S,0)[1]=0.;
306                 ssGetOutputPortRealSignal(S,0)[2]=0.;
307                 ssGetOutputPortRealSignal(S,1)[0]=0.;
308                 printf("switching AMR magnetometer off...\n");
309             }
310             // Switch to power toggle receive mode.
311             SET_MODUS(1);
312             // Waiting is done. Reset counter for next time.
313             SET_WAITING(0);
314             //return;
315         }
316         else
317         {
318             // Loop to wait one second.
319             if (WAITING ==0 ) {printf("Waiting for 1sec before sending toggle command\n
                    ");}
320             TXBytes[0]=0;
321             setNumTXBytes[0]=0;
322             // While waiting flush input line (there shouldn't be anything in it)
323             setNumRXBytes[0]=9;
324             SET_WAITING(WAITING+1);
325         }
326     }
327     // Modus 1: Receive AMR power on/off signal
328     if (MODUS == 1)
329     {
330         setNumRXBytes[0]=1;
331         // 'watchdog' for asumed power toggle.
332         if (ASSUME_SEND > 0)
333         {
334             SET_ASSUME_SEND(ASSUME_SEND+1);
335         }
```

```
336         // Assumed power toggel
337         if (ASSUME_SEND * ssGetSampleTime(S,0) >1.)
338         {
339             SET_ASSUME_SEND(0);
340             SET_AMRSTATUS(0);
341             SET_STARTUP(0);
342             SET_MODUS(2);
343             printf("Assuming power toggle.\n");
344         }
345         // Is the command send properly, then stop sending the command.
346         if (*NumTXBytes == 1)
347         {
348             TXBytes[0]=0;
349             setNumTXBytes[0]=0;
350             SET_ASSUME_SEND(1);
351         }
352         // The acknowlege message is only 1 byte long.
353         if (*NumRXBytes == 1)
354         {
355             // Switch through possible acknowlege messages.
356             switch (RXBytes[0])
357             {
358                 case 0x98:
359                     // Set internal reference to power mode.
360                     SET_AMRSTATUS(0);
361                     SET_NUM_BYTES_OF_PACKET(0);
362                     SET_STARTUP(0);
363                     printf("ARM magnetometer status ist now: offline\n");
364                     // Power toggle done. Now we can receive new data.
365                     SET_MODUS(2);
366                     break;
367                 case 0x99:
368                     SET_AMRSTATUS(1);
369                     SET_NUM_BYTES_OF_PACKET(0);
370                     SET_STARTUP(0);
371                     printf("ARM magnetometer status ist now: online\n");
372                     //return;
373                     SET_MODUS(2);
374                     break;
375             }
376         }
377     }
378     // Modus 2: Request data or no data (0 measurements per second)
379     if (MODUS == 2)
380     {
381         // Not waiting(?), next command can be send
382         // Is the AMR analogue part switch on or is it of and no request?
383         if (AMRSTATUS==1 || (AMRSTATUS==0 && *measurementspersecond==0))
384         {
385             // How fast do i want to get a result, or how often per second?
386             switch (*measurementspersecond)
387             {
388                 case 1:
389                     // For checking later whether the righ data was send, or not.
390                     SET_REQUESTED_DATA(0x50);
391                     // Set byte to request data.
392                     TXBytes[0]=0x50;
393                     // The command to request data is 1 byte long.
394                     setNumTXBytes[0]=1;
395                     // The answer will be 9 byte long
396                     setNumRXBytes[0]=9;
397                     // Switch to receive modus.
```

```
398                     SET_MODUS(3);
399                     break;
400                 case 2:
401                     SET_REQUESTED_DATA(0x51);
402                     TXBytes[0]=0x51;
403                     setNumTXBytes[0]=1;
404                     setNumRXBytes[0]=9;
405                     SET_MODUS(3);
406                     break;
407                 case 4:
408                     SET_REQUESTED_DATA(0x52);
409                     TXBytes[0]=0x52;
410                     setNumTXBytes[0]=1;
411                     setNumRXBytes[0]=9;
412                     SET_MODUS(3);
413                     break;
414                 case 6:
415                     SET_REQUESTED_DATA(0x53);
416                     TXBytes[0]=0x53;
417                     setNumTXBytes[0]=1;
418                     setNumRXBytes[0]=9;
419                     SET_MODUS(3);
420                     break;
421                 // default means no request
422                 default:
423                     SET_REQUESTED_DATA(0x00);
424                     TXBytes[0]=0;
425                     setNumTXBytes[0]=0;
426                     setNumRXBytes[0]=0;
427                     // Output should be zero when no data is requested anymore.
428                     ssGetOutputPortRealSignal(S,0)[0]=0.;
429                     ssGetOutputPortRealSignal(S,0)[1]=0.;
430                     ssGetOutputPortRealSignal(S,0)[2]=0.;
431                     ssGetOutputPortRealSignal(S,1)[0]=0.;
432             }
433         }
434         else
435         {
436             //printf("Cannot request data from offline AMR analogue part, error!\n");
437             // If analogue part of AMR magnetometer is switch of,
438             // nothing will be requested and the output signal should be zero.
439             TXBytes[0]=0;
440             setNumTXBytes[0]=0;
441             ssGetOutputPortRealSignal(S,0)[0]=0.;
442             ssGetOutputPortRealSignal(S,0)[1]=0.;
443             ssGetOutputPortRealSignal(S,0)[2]=0.;
444             ssGetOutputPortRealSignal(S,1)[0]=0.;
445         }
446     }
447     #endif /* MATLAB_MEX_FILE */
448 }
449
450
451
452 #undef MDL_UPDATE  /* Change to #undef to remove function */
453 #if defined(MDL_UPDATE)
454   /* Function: mdlUpdate ======================================================
455    * Abstract:
456    *    This function is called once for every major integration time step.
457    *    Discrete states are typically updated here, but this function is useful
458    *    for performing any tasks that should only take place once per
459    *    integration step.
```

```
460     */
461    static void mdlUpdate(SimStruct *S, int_T tid)
462    {
463    }
464  #endif /* MDL_UPDATE */
465
466
467
468  #undef MDL_DERIVATIVES  /* Change to #undef to remove function */
469  #if defined(MDL_DERIVATIVES)
470    /* Function: mdlDerivatives =============================================
471     * Abstract:
472     *    In this function, you compute the S-function block's derivatives.
473     *    The derivatives are placed in the derivative vector, ssGetdX(S).
474     */
475    static void mdlDerivatives(SimStruct *S)
476    {
477    }
478  #endif /* MDL_DERIVATIVES */
479
480
481
482  /* Function: mdlTerminate =============================================
483   * Abstract:
484   *    In this function, you should perform any actions that are necessary
485   *    at the termination of a simulation.  For example, if memory was
486   *    allocated in mdlStart, this is the place to free it.
487   */
488  static void mdlTerminate(SimStruct *S)
489  {
490      #ifndef MATLAB_MEX_FILE
491    // Free stuff
492    free(PACKET_BUFFER);
493      #endif /* MATLAB_MEX_FILE */
494  }
495
496
497  /*=============================================================*
498   * See sfuntmpl_doc.c for the optional S-function methods *
499   *=============================================================*/
500
501  /*=============================*
502   * Required S-function trailer *
503   *=============================*/
504
505  #ifdef  MATLAB_MEX_FILE    /* Is this file being compiled as a MEX-file? */
506  #include "simulink.c"      /* MEX-file interface mechanism */
507  #else
508  #include "cg_sfun.h"       /* Code generation registration function */
509  #endif
```

C.4 TMCM-310 stepper control board

C.4.1 Simulink™ block

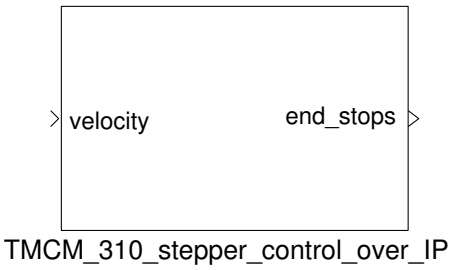


Figure C.4: Simulink™ block of the software interface for the TMCM-310 stepper motor control board

C.4.2 Source code

Listing C.4: HWI_TMCM_310_sf.c

```
1  #define S_FUNCTION_NAME  HWI_TMCM_310_sf
2  #define S_FUNCTION_LEVEL 2
3
4  /*
5   * Need to include simstruc.h for the definition of the SimStruct and
6   * its associated macro definitions.
7   */
8  #include "simstruc.h"
9
10 #define MAXRXTXBYTES (ssGetSFcnParam(S,0))
11
12 // DEBUG
13 //#define DEBUG 1
14 //-----
15 #ifndef MATLAB_MEX_FILE
16 // include files here
17 #endif
18
19 static void mdlInitializeSizes(SimStruct *S)
20 {
21     int maxrxtxbytes = (int) mxGetScalar(MAXRXTXBYTES);
22     // Parameters from the mask
23     ssSetNumSFcnParams(S, 1);  /* Number of expected parameters: maxrxtxbytes*/
24
25     if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) {
26         /* Return if number of expected != number of actual parameters */
27         return;
28     }
29     // 4 input ports:
30     if (!ssSetNumInputPorts(S,4)) return;
31     /*Input Port 0: velocity vector for steppers */
32     ssSetInputPortWidth(S, 0,3);
33     ssSetInputPortDataType(S, 0, SS_DOUBLE);
34     ssSetInputPortComplexSignal(S,  0, COMPLEX_NO);
```

```
35      ssSetInputPortRequiredContiguous(S, 0, true); /*direct input signal access*/
36      ssSetInputPortDirectFeedThrough(S, 0, 1);
37      /*Input Port 1: numRXBytes */
38      ssSetInputPortWidth(S, 1, 1);
39      ssSetInputPortDataType(S, 1, SS_UINT32);
40      ssSetInputPortComplexSignal(S,  1, COMPLEX_NO);
41      ssSetInputPortRequiredContiguous(S, 1, true); /*direct input signal access*/
42      ssSetInputPortDirectFeedThrough(S, 1, 1);
43      /*Input Port 2: RXbytes */
44      ssSetInputPortWidth(S, 2, maxrxtxbytes);
45      ssSetInputPortDataType(S, 2, SS_UINT8);
46      ssSetInputPortComplexSignal(S,  2, COMPLEX_NO);
47      ssSetInputPortRequiredContiguous(S, 2, true); /*direct input signal access*/
48      ssSetInputPortDirectFeedThrough(S, 2, 1);
49      /*Input Port 3: numTXBytes */
50      ssSetInputPortWidth(S, 3, 1);
51      ssSetInputPortDataType(S, 3, SS_UINT32);
52      ssSetInputPortComplexSignal(S,  3, COMPLEX_NO);
53      ssSetInputPortRequiredContiguous(S, 3, true); /*direct input signal access*/
54      ssSetInputPortDirectFeedThrough(S, 3, 1);
55
56      // 4 output ports:
57      if (!ssSetNumOutputPorts(S, 4)) return;
58      /* Output Port 0: reqnumRXBytes */
59      ssSetOutputPortWidth(S, 0, 1);
60      ssSetOutputPortDataType(S, 0, SS_UINT32);
61      ssSetOutputPortComplexSignal(S, 0, COMPLEX_NO);
62      /* Output Port 1: reqnumTXBytes */
63      ssSetOutputPortWidth(S, 1, 1);
64      ssSetOutputPortDataType(S, 1, SS_UINT32);
65      ssSetOutputPortComplexSignal(S, 1, COMPLEX_NO);
66      /* Output Port 2: TXBytes */
67      ssSetOutputPortWidth(S, 2, maxrxtxbytes);
68      ssSetOutputPortDataType(S, 2, SS_UINT8);
69      ssSetOutputPortComplexSignal(S, 2, COMPLEX_NO);
70
71      /* Output Port 2: endstops */
72      ssSetOutputPortWidth(S, 3, 3);
73      ssSetOutputPortDataType(S, 3, SS_INT8);
74      ssSetOutputPortComplexSignal(S, 3, COMPLEX_NO);
75      // global variables:
76      ssSetNumIWork(S, 8); //integer: modus,packet posision,cycle,axis,packet_len,buffer
            position ,found beginpacket,cycletimer
77      ssSetNumPWork(S, 1); // pointer buffer:
78  }
79
80  /* Function: mdlInitializeSampleTimes =====================================
81   * Abstract:
82   *    This function is used to specify the sample time(s) for your
83   *    S-function. You must register the same number of sample times as
84   *    specified in ssSetNumSampleTimes.
85   */
86  static void mdlInitializeSampleTimes(SimStruct *S)
87  {
88      ssSetSampleTime(S, 0, INHERITED_SAMPLE_TIME);
89      ssSetOffsetTime(S, 0, FIXED_IN_MINOR_STEP_OFFSET);
90
91  }
92
93
94
95  #define MDL_INITIALIZE_CONDITIONS   /* Change to #undef to remove function */
```

```
96  #if defined(MDL_INITIALIZE_CONDITIONS)
97    /* Function: mdlInitializeConditions ========================================
98     * Abstract:
99     *    In this function, you should initialize the continuous and discrete
100    *    states for your S-function block.  The initial states are placed
101    *    in the state vector, ssGetContStates(S) or ssGetRealDiscStates(S).
102    *    You can also perform any other initialization activities that your
103    *    S-function may require. Note, this routine will be called at the
104    *    start of simulation and if it is present in an enabled subsystem
105    *    configured to reset states, it will be call when the enabled subsystem
106    *    restarts execution to reset the states.
107    */
108   static void mdlInitializeConditions(SimStruct *S)
109   {
110   }
111 #endif /* MDL_INITIALIZE_CONDITIONS */
112
113 #define SET_MODUS(a) ssSetIWorkValue(S,0,a)
114 #define MODUS ssGetIWorkValue(S,0)
115
116 #define SET_PACKET_POSITION(a) ssSetIWorkValue(S,1,a)
117 #define PACKET_POSITION ssGetIWorkValue(S,1)
118
119 #define SET_CYCLE(a) ssSetIWorkValue(S,2,a)
120 #define CYCLE ssGetIWorkValue(S,2)
121
122 #define SET_AXIS(a) ssSetIWorkValue(S,3,a)
123 #define AXIS ssGetIWorkValue(S,3)
124
125 #define SET_PACKET_LEN(a) ssSetIWorkValue(S,4,a)
126 #define PACKET_LEN ssGetIWorkValue(S,4)
127
128 #define SET_BUFFER_POSITION(a) ssSetIWorkValue(S,5,a)
129 #define BUFFER_POSITION ssGetIWorkValue(S,5)
130
131 #define SET_BEGIN_PACKET(a) ssSetIWorkValue(S,6,a)
132 #define BEGIN_PACKET ssGetIWorkValue(S,6)
133
134 #define SET_CYCLE_TIMER(a) ssSetIWorkValue(S,7,a)
135 #define CYCLE_TIMER ssGetIWorkValue(S,7)
136
137 #define SET_BUFFER(a) ssSetPWorkValue(S,0,a)
138 #define BUFFER ssGetPWorkValue(S,0)
139
140
141
142 #define MDL_START  /* Change to #undef to remove function */
143 #if defined(MDL_START)
144   /* Function: mdlStart =======================================================
145    * Abstract:
146    *    This function is called once at start of model execution. If you
147    *    have states that should be initialized once, this is the place
148    *    to do it.
149    */
150   static void mdlStart(SimStruct *S)
151   {
152     #ifndef MATLAB_MEX_FILE
153     // max. width of transmission vector
154     int maxrxtxbytes = (int) mxGetScalar(MAXRXTXBYTES);
155     // start with modus 0
156     SET_MODUS(0);
157     // initialize things
```

```
158     SET_PACKET_POSITION(0);
159     SET_CYCLE(0);
160     SET_AXIS(0);
161     // receive buffer
162     char * buffer=malloc(sizeof(char)*maxrxtxbytes);
163     bzero(buffer,maxrxtxbytes);
164     SET_BUFFER(buffer);
165     SET_BUFFER_POSITION(0);
166     SET_BEGIN_PACKET(0);
167     // debug timer.
168     SET_CYCLE_TIMER(0);
169     #endif /* MATLAB_MEX_FILE */
170   }
171 #endif /*  MDL_START */
172
173
174
175 /* Function: mdlOutputs =======================================================
176  * Abstract:
177  *    In this function, you compute the outputs of your S-function
178  *    block.
179  */
180 static void mdlOutputs(SimStruct *S, int_T tid)
181 {
182     #ifndef MATLAB_MEX_FILE
183     // Input port signals
184     const double    *velocity  = (const double *)  ssGetInputPortSignal(S,0);
185     const uint32_T   *numRXBytes  = (const uint32_T *)  ssGetInputPortSignal(S,1);
186     const uint8_T   *RXBytes = (const uint8_T *)  ssGetInputPortSignal(S,2);
187     const uint32_T   *numTXBytes  = (const uint32_T *)  ssGetInputPortSignal(S,3);
188     //-------------------------------------------------------------------------//
189     //-------------------------------------------------------------------------//
190     // Output port signals
191     uint32_T *reqNumRXBytes = (uint32_T *)  ssGetOutputPortRealSignal(S,0);
192     uint32_T *reqNumTXBytes = (uint32_T *)  ssGetOutputPortRealSignal(S,1);
193     uint8_T *TXBytes = (uint8_T *)  ssGetOutputPortRealSignal(S,2);
194     int8_T *endstops = (uint8_T *)  ssGetOutputPortRealSignal(S,3);
195
196     #ifdef DEBUG
197         // Debug mode, time between commands (full run: through 3 cycles and 3 axis)
198         if (MODUS==0 && CYCLE==0 && AXIS==0 && PACKET_POSITION==0)
199         {
200             printf("Cycle time is: %f sec\n",ssGetSampleTime(S,0)*CYCLE_TIMER);
201             SET_CYCLE_TIMER(0);
202
203         }
204         SET_CYCLE_TIMER(CYCLE_TIMER+1);
205     #endif
206     // Send modus
207     if (MODUS == 0)
208     {
209         // Cycle through different commands: get information about left and right end
                stop and commadning rotation velocities.
210         switch (CYCLE)
211         {
212             case 0:
213             {
214                 // get ref swith left
215                 // Build standard packet with variable AXIS
216                 char gap9_packet[]={'A','G','A','P',' ','9',',',AXIS+48,13};
217                 // Set packet lenght
218                 SET_PACKET_LEN(9);
```

```
219                 // Set one transmission byte (PACKET_POSITION) from packet. Every byte
                        has to be send seperatly and is echod back from stepper system.
220                 TXBytes[0]=gap9_packet[PACKET_POSITION];
221                 // send exactly one byte
222                 reqNumTXBytes[0]=1;
223                 // receive 20, does not matter. Suffice for all packets, smaller values
                        possible.
224                 reqNumRXBytes[0]=20;
225                 // Set one byte on output now wait in modus 1 for reply,
226                 SET_MODUS(1);
227                 break;
228             }
229             case 1:
230             {
231                 // get ref swith right
232                 // almost same as above...
233                 char gap10_packet[]={'A','G','A','P',' ','1','0',',',AXIS+48,13};
234                 SET_PACKET_LEN(10);
235                 TXBytes[0]=gap10_packet[PACKET_POSITION];
236                 reqNumTXBytes[0]=1;
237                 reqNumRXBytes[0]=20;
238                 SET_MODUS(1);
239                 break;
240             }
241             case 2:
242             {
243                 // ROL or ROR
244                 // Rotate a stepper with constant speed to left (roll left ROL) or
                        right (roll right ROR).
245                 char direction;
246                 int velo;
247                 // If positive velocity -> ROR
248                 if (velocity[AXIS] > 0 )
249                 {
250                     direction='R';
251                     velo=(int) velocity[AXIS];
252                 }
253                 else // IF negetive velocity -> ROL
254                 {
255                     direction='L';
256                     velo=-1* ((int) velocity[AXIS]);
257                 }
258                 // Set at most maximum velocity.
259                 if (velo > 2047) velo=2047;
260                 // Calculate a four digit integer to four chars
261                 char a,b,c,d;
262                 a= velo/1000+48;
263                 b= (velo%1000)/100+48;
264                 c= ((velo%1000)%100)/10+48;
265                 d= (((velo%1000)%100)%10)+48;
266                 // Build packet for ROL or ROR.
267                 char RO_packet[]={'A','R','O',direction,' ',AXIS+48,',',a,b,c,d,13};
268                 SET_PACKET_LEN(12);
269                 TXBytes[0]=RO_packet[PACKET_POSITION];
270                 reqNumTXBytes[0]=1;
271                 reqNumRXBytes[0]=20;
272                 SET_MODUS(1);
273                 break;
274             }
275         }
276
277     }
```

```
278     //receive modus
279     if (MODUS == 1)
280     {
281         // When in Modus 1, one byte had to be transmitted
282         if ((int)numTXBytes[0] > 0)
283         {
284             // if it was send, stop repeating!
285             reqNumTXBytes[0]=0;
286         }
287         // If something was received
288         if ((int)numRXBytes[0] > 0)
289         {
290             // Was the complete packet send one byte by another?
291             if (PACKET_POSITION + 1 >= PACKET_LEN)
292             {
293                 char * buffer=BUFFER;
294                 int i=0,process_packet=0;
295                 // Go through all received bytes.
296                 for(i=0;i < numRXBytes[0];i++)
297                 {
298                     // Write bytes in buffer
299                     buffer[i+BUFFER_POSITION]=RXBytes[i];
300                     // Find beginning of the reply packet by character 'B'.
301                     if (buffer[i+BUFFER_POSITION]=='B') SET_BEGIN_PACKET(1);
302                     // IF carriage return is in received byte and beginning of packet
                            was already found.
303                     if (buffer[i+BUFFER_POSITION]==13 && BEGIN_PACKET==1)
                            process_packet=1;
304                 }
305                 // Set new buffer position if beginning end ending was not found in
                        received packet (waiting until everything is received)
306                 SET_BUFFER_POSITION(i+BUFFER_POSITION);
307                     // If complete packet in buffer
308                     if (process_packet==1)
309                     {
310                         // 'BA 100' in received data means everything is ok, following
                                this a value could be included (infront of BA is the last
                                echo of chr(13) 100 CR)
311                         if (buffer[4] == '1' && buffer[5] == '0' && buffer[6] == '0')
312                         {
313                             int cycleall=0;
314                             // What meaning does the reply have?
315                             if (CYCLE==0)
316                             {
317                                 // Reply for left reference switch: Switch active 1, or
                                        inactive
318                                 if (buffer[8]=='0' || buffer[8]=='1')
319                                 {
320                                   // Set end stop output. Careful: Only set to zero if
                                          left switch was -1
321                                   if (buffer[8]=='1') endstops[AXIS]=(buffer[8]-48)*-1;
322                                   else if (buffer[8]=='0' && endstops[AXIS]==-1)
                                          endstops[AXIS]=0;
323                                   // packet was received an everything was ok with it.
                                          -> next command and or axis.
324                                   cycleall=1;
325                                 }
326                             }
327                             if (CYCLE==1)
328                             {
329                                 // As above for right reference switch.
330                                 if (buffer[8]=='0' || buffer[8]=='1')
```

```
331                                 {
332                                     if (buffer[8]=='1') endstops[AXIS]=(buffer[8]-48);
333                                     else if (buffer[8]=='0' && endstops[AXIS]==1)
                                            endstops[AXIS]=0;
334                                     cycleall=1;
335                                 }
336                             }
337                             if (CYCLE==2)
338                             {
339                                 // The velocity was changed. Dont care what it says, as
                                        long as it said "BA 100".
340                                 //ROL or ROR
341                                 cycleall=1;
342                             }
343                             // Cycle through CYCLE(0=left switch,1=right switch,2=set
                                    velocity) and AXIS.(0=x,1=y,2=z)
344                             if (cycleall==1)
345                             {
346                                 SET_AXIS(AXIS+1);
347                                 if (AXIS > 2)
348                                 {
349                                     SET_AXIS(0);
350                                     SET_CYCLE(CYCLE+1);
351                                     if (CYCLE > 2) SET_CYCLE(0);
352                                 }
353                             }
354                         }
355                         // Complete packet was received. Go back to sending modus for a
                                new request, possible if something went wrong the same
                                packet could be resent.
356                         SET_MODUS(0);
357                         // Clear buffer
358                         int maxrxtxbytes = (int) mxGetScalar(MAXRXTXBYTES);
359                         bzero(buffer,maxrxtxbytes);
360                         // Reset variables
361                         SET_BEGIN_PACKET(0);
362                         SET_BUFFER_POSITION(0);
363                         SET_PACKET_POSITION(0);
364                     }
365             }
366             else
367             {
368                 // The echo from the Commandbyte was received, increase packet position
                        , so that next byte in packet can be sent.
369                 SET_PACKET_POSITION(PACKET_POSITION + 1);
370                 // Go back to sending modus.
371                 SET_MODUS(0);
372             }
373
374         }
375     }
376     #endif /* MATLAB_MEX_FILE */
377 }
378
379
380
381 #undef MDL_UPDATE  /* Change to #undef to remove function */
382 #if defined(MDL_UPDATE)
383   /* Function: mdlUpdate ======================================================
384    * Abstract:
385    *    This function is called once for every major integration time step.
386    *    Discrete states are typically updated here, but this function is useful
```

```
387     *    for performing any tasks that should only take place once per
388     *    integration step.
389     */
390    static void mdlUpdate(SimStruct *S, int_T tid)
391    {
392    }
393  #endif /* MDL_UPDATE */
394
395
396
397  #undef MDL_DERIVATIVES  /* Change to #undef to remove function */
398  #if defined(MDL_DERIVATIVES)
399    /* Function: mdlDerivatives =============================================
400     * Abstract:
401     *    In this function, you compute the S-function block's derivatives.
402     *    The derivatives are placed in the derivative vector, ssGetdX(S).
403     */
404    static void mdlDerivatives(SimStruct *S)
405    {
406    }
407  #endif /* MDL_DERIVATIVES */
408
409
410
411  /* Function: mdlTerminate =================================================
412   * Abstract:
413   *    In this function, you should perform any actions that are necessary
414   *    at the termination of a simulation.  For example, if memory was
415   *    allocated in mdlStart, this is the place to free it.
416   */
417  static void mdlTerminate(SimStruct *S)
418  {
419      #ifndef MATLAB_MEX_FILE
420      // Free stuff
421      free(BUFFER);
422      #endif /* MATLAB_MEX_FILE */
423  }
424
425
426  /*=====================================================*
427   * See sfuntmpl_doc.c for the optional S-function methods *
428   *=====================================================*/
429
430  /*=============================*
431   * Required S-function trailer *
432   *=============================*/
433
434  #ifdef  MATLAB_MEX_FILE    /* Is this file being compiled as a MEX-file? */
435  #include "simulink.c"      /* MEX-file interface mechanism */
436  #else
437  #include "cg_sfun.h"       /* Code generation registration function */
438  #endif
```

C.5 TCP/IP socket interface

C.5.1 Simulink™ block

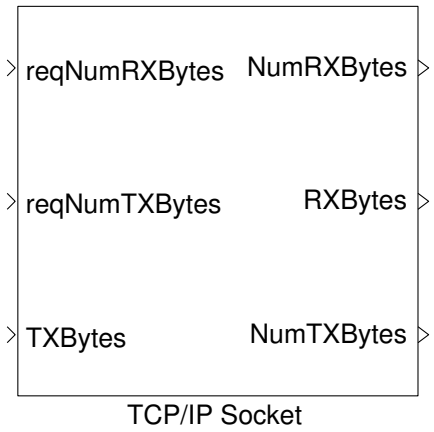


Figure C.5: Simulink™ block of the TCP/IP socket interface

C.5.2 Source code

Listing C.5: HWI_socket_sf.c

```
1  #define S_FUNCTION_NAME  HWI_socket_sf
2  #define S_FUNCTION_LEVEL 2
3
4  /*
5      IMPORTANT: It is not possible to have a server and a client in the
6      same simulink model. they are in the same thread so no connection
7      process is possible. Two different models with one a server and one
8      as client works.
9      The listen/connect/accept part is BLOCKING the thread!
10     Multiple socket user or creator in one model could be possible, if
11     the connect or accept connection from outside the thread. The order in
12     which this happens is done by simulink.
13
14     The source for this is mainly taken from example from the internet.
15     Google somthing like: "socket programming", "linux network socket",
16     "posix network socket", "bsd sockets"
17 */
18
19 /*
20  * Need to include simstruc.h for the definition of the SimStruct and
21  * its associated macro definitions.
22  */
23 #include "simstruc.h"
24 #define SOCKETTYPE (ssGetSFcnParam(S,0)) //Server/Client
25 #define SERVER (ssGetSFcnParam(S,1)) //server aderss ip or hostname
26 #define PORT (ssGetSFcnParam(S,2)) // port numer
27 #define MAXRXTXBYTES (ssGetSFcnParam(S,3)) //Maximal data vector width.
28
29 //#define MAXPORTS_RC 11
```

```
30
31 #ifndef MATLAB_MEX_FILE
32
33 #include <stdio.h>
34 #include <stdlib.h>
35 #include <unistd.h>
36 #include <string.h>
37 #include <sys/types.h>
38 #include <sys/socket.h>
39 #include <netinet/in.h>
40 #include <netdb.h>
41 #include <fcntl.h>
42
43 #endif
44
45 static void mdlInitializeSizes(SimStruct *S)
46 {
47     // Maximal data vector width.
48     int maxrxtxbytes = (int) mxGetScalar(MAXRXTXBYTES);
49     // Parameters from the mask
50     ssSetNumSFcnParams(S, 4);  /* Number of expected parameters: thesockettype,
          theserver,theport,maxrxtxbytes */
51     if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) {
52         /* Return if number of expected != number of actual parameters */
53         return;
54     }
55
56     //-----------------------------------------------------------------------//
57     // Input ports: 3
58     if (!ssSetNumInputPorts(S, 3)) return;
59     /*Input Port 0: reqNumRXBytes */
60     ssSetInputPortWidth(S, 0,1);
61     ssSetInputPortDataType(S, 0, SS_UINT32);
62     ssSetInputPortComplexSignal(S,  0, COMPLEX_NO);
63     ssSetInputPortRequiredContiguous(S, 0, true); /*direct input signal access*/
64     ssSetInputPortDirectFeedThrough(S, 0, 1);
65     /*Input Port 1: reqNumTXBytes */
66     ssSetInputPortWidth(S, 1,1);
67     ssSetInputPortDataType(S, 1, SS_UINT32);
68     ssSetInputPortComplexSignal(S,  1, COMPLEX_NO);
69     ssSetInputPortRequiredContiguous(S, 1, true); /*direct input signal access*/
70     ssSetInputPortDirectFeedThrough(S, 1, 1);
71     /*Input Port 2: txbytes */
72     ssSetInputPortWidth(S, 2,maxrxtxbytes);
73     ssSetInputPortDataType(S, 2, SS_UINT8);
74     ssSetInputPortComplexSignal(S,  2, COMPLEX_NO);
75     ssSetInputPortRequiredContiguous(S, 2, true); /*direct input signal access*/
76     ssSetInputPortDirectFeedThrough(S, 2, 1);
77
78     //-----------------------------------------------------------------------//
79     //-----------------------------------------------------------------------//
80     // Output ports ports: 3
81     if (!ssSetNumOutputPorts(S, 3)) return;
82     /* Output Port 0: numRXBytes */
83     ssSetOutputPortWidth(S, 0, 1);
84     ssSetOutputPortDataType(S, 0, SS_UINT32);
85     ssSetOutputPortComplexSignal(S, 0, COMPLEX_NO);
86     /* Output Port 1: RXBytes */
87     ssSetOutputPortWidth(S, 1, maxrxtxbytes);
88     ssSetOutputPortDataType(S, 1, SS_UINT8);
89     ssSetOutputPortComplexSignal(S, 1, COMPLEX_NO);
90     /* Output Port 2: numTXBytes */
```

```
91      ssSetOutputPortWidth(S, 2, 1);
92      ssSetOutputPortDataType(S, 2, SS_UINT32);
93      ssSetOutputPortComplexSignal(S, 2, COMPLEX_NO);
94
95      // global variables:
96      ssSetNumIWork(S, 3); //integer: SOCKFD, newsockfd, connect client:
97      ssSetNumPWork(S, 1); // pointer:sockfd
98  }
99
100
101
102
103 /* Function: mdlInitializeSampleTimes =========================================
104  * Abstract:
105  *    This function is used to specify the sample time(s) for your
106  *    S-function. You must register the same number of sample times as
107  *    specified in ssSetNumSampleTimes.
108  */
109 static void mdlInitializeSampleTimes(SimStruct *S)
110 {
111     ssSetSampleTime(S, 0, INHERITED_SAMPLE_TIME);
112     ssSetOffsetTime(S, 0, FIXED_IN_MINOR_STEP_OFFSET);
113
114 }
115
116
117
118 #define MDL_INITIALIZE_CONDITIONS   /* Change to #undef to remove function */
119 #if defined(MDL_INITIALIZE_CONDITIONS)
120   /* Function: mdlInitializeConditions ========================================
121    * Abstract:
122    *    In this function, you should initialize the continuous and discrete
123    *    states for your S-function block.  The initial states are placed
124    *    in the state vector, ssGetContStates(S) or ssGetRealDiscStates(S).
125    *    You can also perform any other initialization activities that your
126    *    S-function may require. Note, this routine will be called at the
127    *    start of simulation and if it is present in an enabled subsystem
128    *    configured to reset states, it will be call when the enabled subsystem
129    *    restarts execution to reset the states.
130    */
131   static void mdlInitializeConditions(SimStruct *S)
132   {
133   }
134 #endif /* MDL_INITIALIZE_CONDITIONS */
135
136 #define SET_SOCKFD(a) ssSetIWorkValue(S,0,a)
137 #define SOCKFD ssGetIWorkValue(S,0)
138 #define SET_OLDSOCKFD(a) ssSetIWorkValue(S,1,a)
139 #define OLDSOCKFD ssGetIWorkValue(S,1)
140
141 #define SET_CONNECT(a) ssSetIWorkValue(S,2,a)
142 #define CONNECT ssGetIWorkValue(S,2)
143
144 #define MDL_START  /* Change to #undef to remove function */
145 #if defined(MDL_START)
146   /* Function: mdlStart =======================================================
147    * Abstract:
148    *    This function is called once at start of model execution. If you
149    *    have states that should be initialized once, this is the place
150    *    to do it.
151    */
152   static void mdlStart(SimStruct *S)
```

```
153   {
154     #ifndef MATLAB_MEX_FILE
155     int maxrxtxbytes = (int) mxGetScalar(MAXRXTXBYTES);
156     int sockettype=mxGetScalar(SOCKETTYPE);
157     printf("MDLSTART: %d\n",sockettype);
158     SET_SOCKFD(0);
159   if (sockettype==1)
160     {
161         // If this is a server:
162         printf("IAMA server!\n");
163         int sockfd, portno;
164         char buffer[256];
165         struct sockaddr_in serv_addr;
166         int n;
167         char port_str[6];
168         bzero(port_str,6);
169         sockfd = socket(AF_INET, SOCK_STREAM, 0);
170         if (sockfd < 0) printf("ERROR opening socket\n");
171         bzero((char *) &serv_addr, sizeof(serv_addr));
172         if(mxGetString(PORT, port_str, sizeof(port_str)-1)!=0)
173             {
174                 printf("Error: could not get port from parameters.\n");
175             }
176         printf("portstr: '%s'\n",port_str);
177         portno = atoi(port_str);
178         serv_addr.sin_family = AF_INET;
179         serv_addr.sin_addr.s_addr = INADDR_ANY;
180         serv_addr.sin_port = htons(portno);
181         if (bind(sockfd, (struct sockaddr *) &serv_addr,
182             sizeof(serv_addr)) < 0)
183             printf("ERROR on binding\n");
184         struct sockaddr_in cli_addr;
185         socklen_t clilen;
186         int newsockfd;
187         listen(sockfd,5);
188         clilen = sizeof(cli_addr);
189         newsockfd = accept(sockfd,
190             (struct sockaddr *) &cli_addr,
191             &clilen);
192         if (newsockfd < 0) printf("ERROR on accept\n");
193         fcntl(newsockfd,F_SETFL,O_NONBLOCK);
194         SET_OLDSOCKFD(sockfd);
195         SET_SOCKFD(newsockfd);
196
197     }
198     else
199     {
200         // If this is a client:
201         printf("IAMA client\n");
202         int sockfd, portno, n;
203         struct sockaddr_in serv_addr;
204         struct hostent *server;
205
206         char buffer[maxrxtxbytes];
207         char port_str[6],server_str[25];
208         bzero(port_str,6);
209         bzero(server_str,25);
210         if(mxGetString(SERVER, server_str, sizeof(server_str)-1)!=0)
211         {
212             printf("Error: could not get server from parameters.\n");
213         }
214         printf("serverstr: '%s'\n",server_str);
```

```
215         if(mxGetString(PORT, port_str, sizeof(port_str)-1)!=0)
216         {
217             printf("Error: could not get port from parameters.\n");
218         }
219         printf("portstr: '%s'\n",port_str);
220         portno = atoi(port_str);
221         if (SOCKFD ==0)
222         {
223             sockfd = socket(AF_INET, SOCK_STREAM, 0);
224         }
225         else
226         {
227             sockfd=SOCKFD;
228         }
229         if (sockfd < 0) printf("ERROR opening socket\n");
230         server = gethostbyname(server_str);
231         if (server == NULL) {
232             fprintf(stderr,"ERROR, no such host\n");
233             exit(0);
234         }
235         bzero((char *) &serv_addr, sizeof(serv_addr));
236         serv_addr.sin_family = AF_INET;
237         bcopy((char *)server->h_addr,
238             (char *)&serv_addr.sin_addr.s_addr,
239             server->h_length);
240         serv_addr.sin_port = htons(portno);
241 
242         if (connect(sockfd,(struct sockaddr *) &serv_addr,sizeof(serv_addr)) < 0)
243             printf("ERROR connecting\n");
244         fcntl(sockfd,F_SETFL,O_NONBLOCK);
245         SET_SOCKFD(sockfd);
246     }
247 
248     #endif /* MATLAB_MEX_FILE */
249   }
250 #endif /*  MDL_START */
251 
252 
253 
254 /* Function: mdlOutputs =======================================================
255  * Abstract:
256  *    In this function, you compute the outputs of your S-function
257  *    block.
258  */
259 static void mdlOutputs(SimStruct *S, int_T tid)
260 {
261     #ifndef MATLAB_MEX_FILE
262     //------------------------------------------------------------------------//
263     // Input port signals
264     const uint32_T  *reqNumRXBytes  = (const uint32_T *)  ssGetInputPortSignal(S,0);
265     const uint32_T *reqNumTXBytes  = (const uint32_T *)  ssGetInputPortSignal(S,1);
266     const uint8_T *TXBytes  = (const uint8_T *)  ssGetInputPortSignal(S,2);
267     //------------------------------------------------------------------------//
268     // Output port signals
269     uint32_T *NumRXBytes = (uint32_T *)  ssGetOutputPortRealSignal(S,0);
270     uint8_T *RXBytes  = (uint8_T *)  ssGetOutputPortRealSignal(S,1);
271     uint32_T *NumTXBytes = (uint32_T *)  ssGetOutputPortRealSignal(S,2);
272     int maxrxtxbytes =  (int) mxGetScalar(MAXRXTXBYTES);
273     int sockettype=mxGetScalar(SOCKETTYPE);
274     int rx,tx;
275 
276     if (reqNumRXBytes[0] > maxrxtxbytes)
```

```
277     {
278         rx=maxrxtxbytes;
279     }
280     else
281     {
282         rx=reqNumRXBytes[0];
283     }
284     if (reqNumTXBytes[0] > maxrxtxbytes)
285     {
286         tx=maxrxtxbytes;
287     }
288     else
289     {
290         tx=reqNumTXBytes[0];
291     }
292     if (rx > 0)
293     {
294         NumRXBytes[0] = read(SOCKFD,(char *)RXBytes, sizeof(char)*rx);
295         // If waiting for somethin but nothing is received than send 0 back
296         if ((int)NumRXBytes[0] <0)
297         {
298             NumRXBytes[0] =0;
299         }
300     }
301     else
302     {
303         NumRXBytes[0] =0;
304     }
305     if (tx > 0)
306     {
307         NumTXBytes[0] = write(SOCKFD,(char *)TXBytes,sizeof(char)*tx);
308         if ((int)NumTXBytes[0] <0)
309         {
310             NumTXBytes[0] =0;
311         }
312     }
313     else
314     {
315         NumTXBytes[0] =0;
316     }
317     #endif /* MATLAB_MEX_FILE */
318 }
319
320
321
322 #undef MDL_UPDATE  /* Change to #undef to remove function */
323 #if defined(MDL_UPDATE)
324   /* Function: mdlUpdate ======================================================
325    * Abstract:
326    *     This function is called once for every major integration time step.
327    *     Discrete states are typically updated here, but this function is useful
328    *     for performing any tasks that should only take place once per
329    *     integration step.
330    */
331   static void mdlUpdate(SimStruct *S, int_T tid)
332   {
333   }
334 #endif /* MDL_UPDATE */
335
336
337
338 #undef MDL_DERIVATIVES  /* Change to #undef to remove function */
```

```
339 #if defined(MDL_DERIVATIVES)
340   /* Function: mdlDerivatives =============================================
341    * Abstract:
342    *    In this function, you compute the S-function block's derivatives.
343    *    The derivatives are placed in the derivative vector, ssGetdX(S).
344    */
345   static void mdlDerivatives(SimStruct *S)
346   {
347   }
348 #endif /* MDL_DERIVATIVES */
349
350
351
352 /* Function: mdlTerminate =============================================
353  * Abstract:
354  *    In this function, you should perform any actions that are necessary
355  *    at the termination of a simulation.  For example, if memory was
356  *    allocated in mdlStart, this is the place to free it.
357  */
358 static void mdlTerminate(SimStruct *S)
359 {
360     #ifndef MATLAB_MEX_FILE
361     if (mxGetScalar(SOCKETTYPE)==1)
362     {
363         close(OLDSOCKFD);
364     }
365     close(SOCKFD);
366     #endif /* MATLAB_MEX_FILE */
367 }
368
369
370 /*=============================================================*
371  * See sfuntmpl_doc.c for the optional S-function methods *
372  *=============================================================*/
373
374 /*=============================*
375  * Required S-function trailer *
376  *=============================*/
377
378 #ifdef  MATLAB_MEX_FILE    /* Is this file being compiled as a MEX-file? */
379 #include "simulink.c"      /* MEX-file interface mechanism */
380 #else
381 #include "cg_sfun.h"       /* Code generation registration function */
382 #endif
```

D ASCII table

Dec	Hex	Oct	Character	Dec	Hex	Oct	Character
0	0x00	000	NUL	32	0x20	040	SP
1	0x01	001	SOH	33	0x21	041	!
2	0x02	002	STX	34	0x22	042	"’
3	0x03	003	ETX	35	0x23	043	#
4	0x04	004	EOT	36	0x24	044	
5	0x05	005	ENQ	37	0x25	045	%
6	0x06	006	ACK	38	0x26	046	&
7	0x07	007	BEL	39	0x27	047	’
8	0x08	010	BS	40	0x28	050	(
9	0x09	011	TAB	41	0x29	051	)
10	0x0a	012	LF	42	0x2a	052	*
11	0x0b	013	VT	43	0x2b	053	+
12	0x0c	014	FF	44	0x2c	054	,
13	0x0d	015	CR	45	0x2d	055	-
14	0x0e	016	SO	46	0x2e	056	.
15	0x0f	017	SI	47	0x2f	057	/
16	0x10	020	DLE	48	0x30	060	0
17	0x11	021	DC1	49	0x31	061	1
18	0x12	022	DC2	50	0x32	062	2
19	0x13	023	DC3	51	0x33	063	3
20	0x14	024	DC4	52	0x34	064	4
21	0x15	025	NAK	53	0x35	065	5
22	0x16	026	SYN	54	0x36	066	6
23	0x17	027	ETB	55	0x37	067	7
24	0x18	030	CAN	56	0x38	070	8
25	0x19	031	EM	57	0x39	071	9
26	0x1a	032	SUB	58	0x3a	072	:
27	0x1b	033	ESC	59	0x3b	073	;
28	0x1c	034	FS	60	0x3c	074	"<
29	0x1d	035	GS	61	0x3d	075	=
30	0x1e	036	RS	62	0x3e	076	">
31	0x1f	037	US	63	0x3f	077	?

Dec	Hex	Oct	Character	Dec	Hex	Oct	Character
64	0x40	100	@	96	0x60	140	‘
65	0x41	101	A	97	0x61	141	a
66	0x42	102	B	98	0x62	142	b
67	0x43	103	C	99	0x63	143	c
68	0x44	104	D	100	0x64	144	d
69	0x45	105	E	101	0x65	145	e
70	0x46	106	F	102	0x66	146	f
71	0x47	107	G	103	0x67	147	g
72	0x48	110	H	104	0x68	150	h
73	0x49	111	I	105	0x69	151	i
74	0x4a	112	J	106	0x6a	152	j
75	0x4b	113	K	107	0x6b	153	k
76	0x4c	114	L	108	0x6c	154	l
77	0x4d	115	M	109	0x6d	155	m
78	0x4e	116	N	110	0x6e	156	n
79	0x4f	117	O	111	0x6f	157	o
80	0x50	120	P	112	0x70	160	p
81	0x51	121	Q	113	0x71	161	q
82	0x52	122	R	114	0x72	162	r
83	0x53	123	S	115	0x73	163	s
84	0x54	124	T	116	0x74	164	t
85	0x55	125	U	117	0x75	165	u
86	0x56	126	V	118	0x76	166	v
87	0x57	127	W	119	0x77	167	w
88	0x58	130	X	120	0x78	170	x
89	0x59	131	Y	121	0x79	171	y
90	0x5a	132	Z	122	0x7a	172	z
91	0x5b	133	[	123	0x7b	173	{
92	0x5c	134	\	124	0x7c	174	|
93	0x5d	135	]	125	0x7d	175	}
94	0x5e	136	^	126	0x7e	176	"
95	0x5f	137	_	127	0x7f	177	DEL

E Project management

Technische Universität Braunschweig

Institut für Luft- und Raumfahrtsysteme

Satellite attitude control system for demonstration purposes

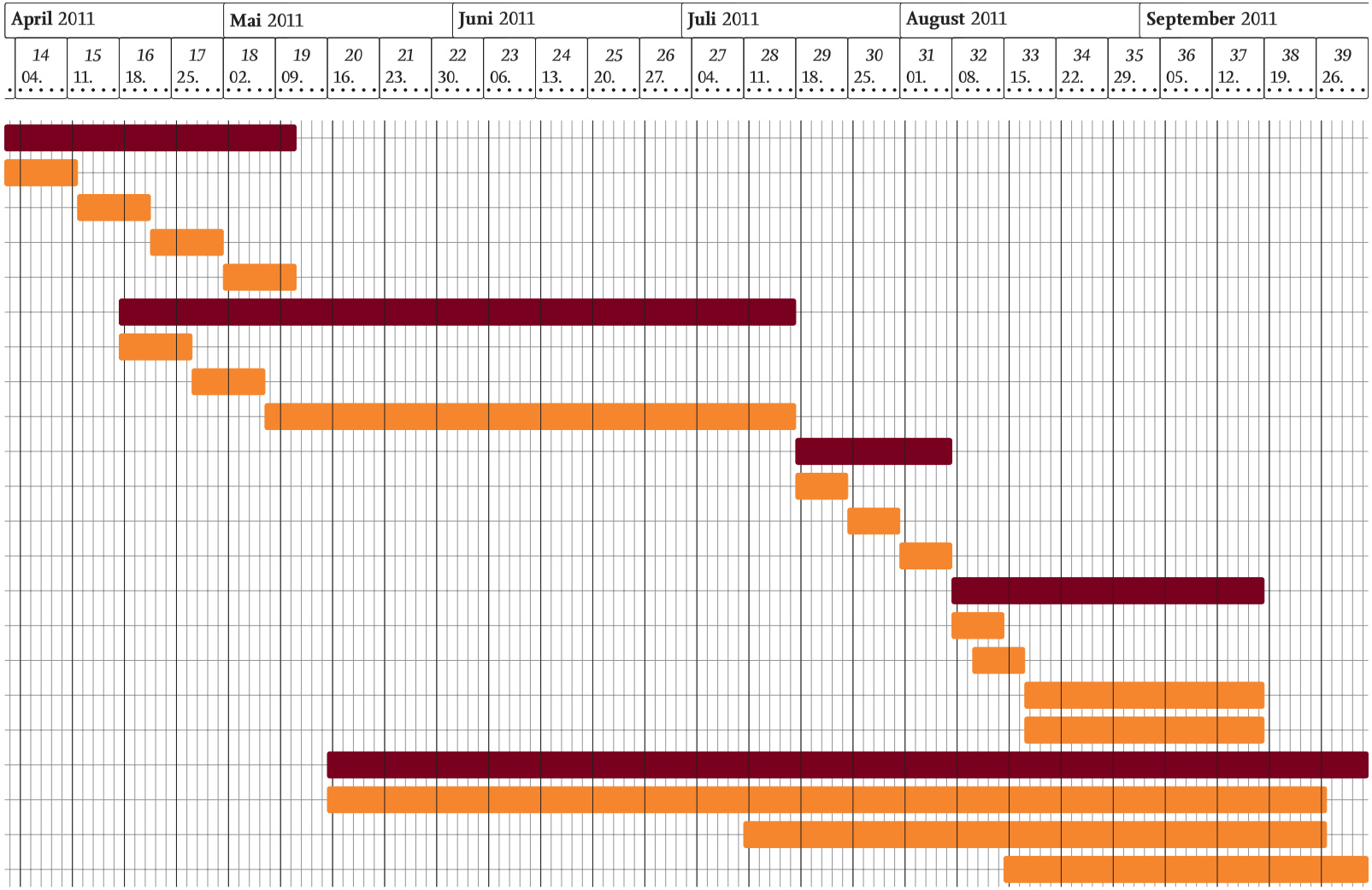
Projektmanagement

25. September 2011

Michael Dumke

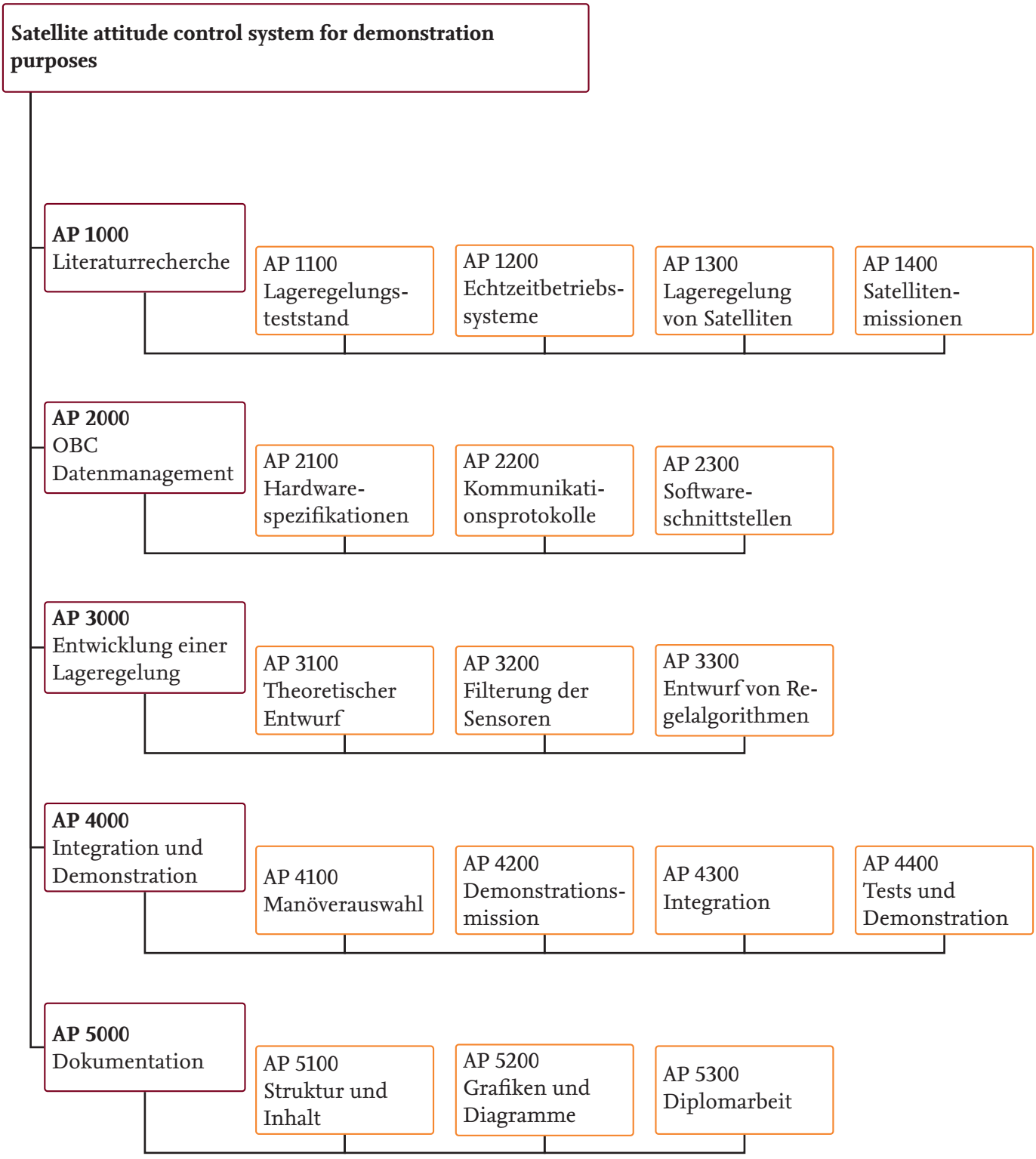
Inhaltsverzeichnis

1 Zeitplanung



Arbeitspaket	Dauer
AP1000 Literaturrecherche	28 Tage
AP1100 Lageregelungsteststand	7 Tage
AP1200 Echtzeitbetriebssysteme	7 Tage
AP1300 Lageregelung von Satelliten	7 Tage
AP1400 Satellitenmissionen	7 Tage
AP2000 OBC Datenmanagement	65 Tage
AP2100 Hardwarespezifikationen	7 Tage
AP2200 Kommunikationsprotokolle	7 Tage
AP2300 Softwareschnittstellen	51 Tage
AP3000 Entwicklung einer Lageregelung	15 Tage
AP3100 Theoretischer Entwurf	5 Tage
AP3200 Filterung der Sensoren	5 Tage
AP3300 Entwurf von Regelalgorithmen	5 Tage
AP4000 Integration und Demonstration	30 Tage
AP4100 Manöverauswahl	5 Tage
AP4200 Demonstrationsmission	5 Tage
AP4300 Integration	23 Tage
AP4400 Tests und Demonstration	23 Tage
AP5000 Dokumentation	100 Tage
AP5100 Struktur und Inhalt	96 Tage
AP5200 Grafiken und Diagramme	56 Tage
AP5300 Diplomarbeit	35 Tage

2 Projektstrukturplan (Work Breakdown Structure)



3 Arbeitspakete

3.1 Literaturrecherche

Satellite attitude control system for demonstration purposes		**AP 1000**	
Titel	Literaturrecherche	**Blatt:**	1 von 22
Verantwortlich	Michael Dumke	**Version:**	1.0
		Datum:	25.09.2011
Start	01.04.2011		
Ende	10.05.2011	**Dauer:**	28 Tage
Bearbeiter	Michael Dumke		

3.1.1 Lageregelungsteststand

Satellite attitude control system for demonstration purposes		**AP 1100**	
Titel	Lageregelungsteststand	**Blatt:**	2 von 22
Verantwortlich	Michael Dumke	**Version:**	1.0
		Datum:	25.09.2011
Start	01.04.2011		
Ende	11.04.2011	**Dauer:**	7 Tage
Bearbeiter	Michael Dumke		

Ziele:

- Überblick über die verwendete Technik der Lageregelungsplattformen des DLRs.

Input:

- Dokumentation und Handbücher der Hard- und Software der Simulationsplattform.
- Dokumentation und Handbücher der Hardware die für die Simulationen verwendet werden sollen (Schwungräder, Magnetometer usw.).

Schnittstellen zu anderen APe:

- AP 2000
- AP 3300
- AP 4000

Aufgaben:

- Lesen von technischen Dokumentationen.

Ergebnisse:

- Erlangung tieferen Verständnis über den gesamten Lageregelungsteststand.
- Aufbau eines ausführlichen Hintergrundwissens für die technische Dokumentation.

3.1.2 Echtzeitbetriebssysteme

Satellite attitude control system for demonstration purposes		**AP 1200**	
Titel	Echtzeitbetriebssysteme	**Blatt:**	3 von 22
Verantwortlich	Michael Dumke	**Version:**	1.0
		Datum:	25.09.2011
Start	12.04.2011		
Ende	20.04.2011	**Dauer:**	7 Tage
Bearbeiter	Michael Dumke		

Ziele:

- Überblick über den Nutzen und die Verwendung von Echtzeitbetriebsystemen, insbesondere QNX™

Input:

- Literatur direkt von QNX™. Insbesondere zum Themengebiet der Seriellenschnittstelle, deren Ansteuerung und Auswertung. Zusätzlich Dokumentation der Hardwarekomponenten, die auf dem Teststand verwendet werden sollen.

Schnittstellen zu anderen APe:

- AP 2200
- AP 2300

Aufgaben:

- Lesen von Büchern

Ergebnisse:

- Erlernen der Fähigkeiten zu Benutzung von insbesondere seriellen Schnittstellen im Rechnerverbund.
- Aufbau eines ausführlichen Hintergrundwissens für die technische Dokumentation.

3.1.3 Lageregelung von Satelliten

Satellite attitude control system for demonstration purposes		**AP 1300**	
Titel	Lageregelung von Satelliten	**Blatt:**	4 von 22
Verantwortlich	Michael Dumke	**Version:**	1.0
		Datum:	25.09.2011
Start	21.04.2011		
Ende	29.04.2011	**Dauer:**	7 Tage
Bearbeiter	Michael Dumke		

Ziele:

- Überblick über die Lageregelung von Satelliten

Input:

- Allgemeine Literatur zum Thema Lageregelung und Reglerauslegung.

Schnittstellen zu anderen APe:

- AP 3000

Aufgaben:

- Lesen von Literatur.

Ergebnisse:

- Erlernen der Fähigkeit zur einfachen Auslegung von Reglern für die Raumfahrt.
- Aufbau eines ausführlichen Hintergrundwissens für die technische Dokumentation.

3.1.4 Satellitenmissionen

Satellite attitude control system for demonstration purposes		**AP 1400**	
Titel	Satellitenmissionen	**Blatt:**	5 von 22
Verantwortlich	Michael Dumke	**Version:**	1.0
		Datum:	25.09.2011
Start	02.05.2011		
Ende	10.05.2011	**Dauer:**	7 Tage
Bearbeiter	Michael Dumke		

Ziele:

- Überblick über elementare Manöver in Raumfahrtmissionen

Input:

- Allgemeine Literatur zum Thema der Raumfahrtmissionen.

Schnittstellen zu anderen APe:

- AP 3300
- AP 4100
- AP 4200

Aufgaben:

- Lesen von Büchern.

Ergebnisse:

- Überblick über grundsätzliche Manöver von Satellitenmissionen, um im späteren Verlauf Beispielmanöver und Szenarien zu entwickeln.

3.2 OBC Datenmanagement

Satellite attitude control system for demonstration purposes		**AP 2000**	
Titel	OBC Datenmanagement	**Blatt:**	6 von 22
Verantwortlich	Michael Dumke	**Version:**	1.0
		Datum:	25.09.2011
Start	18.04.2011		
Ende	15.07.2011	**Dauer:**	65 Tage
Bearbeiter	Michael Dumke		

3.2.1 Hardwarespezifikationen

Satellite attitude control system for demonstration purposes		**AP 2100**	
Titel	Hardwarespezifikationen	**Blatt:**	7 von 22
Verantwortlich	Michael Dumke	**Version:**	1.0
		Datum:	25.09.2011
Start	18.04.2011		
Ende	26.04.2011	**Dauer:**	7 Tage
Bearbeiter	Michael Dumke		

Ziele:

- Aufbau der physikalischen Kommunikation zwischen Sensoren/Aktuatoren und dem On-board Computer.

Input:

- Dokumentation der Hardwarekomponenten (Sensoren / Aktuatoren / On-board Computer)
- Literatur über verschiedene Hardwareschnittstellen, insbesondere serielle Verbindungen.

Schnittstellen zu anderen APe:

- AP 1100
- AP 4300

Aufgaben:

- Aufbau des physikalischen Kommunikationsnetzes auf der Simulationsplattform.
- Herstellung von Kabel- und Steckverbindungen.
- Inbetriebnahme der Hardwareelemente.

Ergebnisse:

- Eine Dateninfrastruktur, die den Austausch von Zuständen über verschiedenste Kommunikationswege zulässt und somit eine zentrale Regelung der Aktuatoren und Sensoren im On-board Computer zulässt.

3.2.2 Kommunikationsprotokolle

Satellite attitude control system for demonstration purposes		**AP 2200**	
Titel	Kommunikationsprotokolle	**Blatt:**	8 von 22
Verantwortlich	Michael Dumke	**Version:**	1.0
		Datum:	25.09.2011
Start	27.04.2011		
Ende	05.05.2011	**Dauer:**	7 Tage
Bearbeiter	Michael Dumke		

Ziele:

- Aufbau der Datenkommunikation zwischen Sensoren/Aktuatoren und dem On-board Computer.

Input:

- Dokumentation der Hardwarekomponenten (Sensoren / Aktuatoren / On-board Computer)
- Literatur über QNX™ oder POSIXartigen Echtzeitbetriebsystemem.

Schnittstellen zu anderen APe:

- AP 1100
- AP 1200
- AP 2300
- AP 4400

Aufgaben:

- Softwareentwicklung von hardwarenahen Treiberinterfaces auf Basis der Kommunikationsprotokolle, die von den Herstellern der Sensorenhardware vorgesehen sind.

Ergebnisse:

- Eine Dateninfrastruktur, die den Austausch von Zuständen über verschiedenste Kommunikationswege zulässt. Diese soll später für die Ansteuerung der Regelungshardware verwendet werden.

3.2.3 Softwareschnittstellen

Satellite attitude control system for demonstration purposes		**AP 2300**	
Titel	Softwareschnittstellen	**Blatt:**	9 von 22
Verantwortlich	Michael Dumke	**Version:**	1.0
		Datum:	25.09.2011
Start	06.05.2011		
Ende	15.07.2011	**Dauer:**	51 Tage
Bearbeiter	Michael Dumke		

Ziele:

- Aufbau von allgemeinen Softwareschnittstellen für die einfache Benutzung auf Anwenderebene.

Input:

- Literatur über QNX™ oder POSIXartigen Betriebsystemem.
- Dokumentationen verschiedener Software, wie z. B. MATLAB™/Simulink™

Schnittstellen zu anderen APe:

- AP 1100
- AP 1200
- AP 2200
- AP 4400

Aufgaben:

- Softwareentwicklung von Interfaces zwischen Betriebsystemebene und Anwendersoftware.
- Implementierung der eigenen Treiber in eine höhere standardisierte Abstraktionsebene zur Vereinfachung der späteren Verwendung ohne genauere Kenntnisse über die hardwarenahe Schicht.

Ergebnisse:

- Leicht zu verwendende sowie standardisierte Kommunikationswege zwischen den Hardwaresensoren und Anwendersoftware.

3.3 Entwicklung einer Lageregelung

Satellite attitude control system for demonstration purposes		**AP 3000**	
Titel	Entwicklung einer Lageregelung	**Blatt:**	10 von 22
Verantwortlich	Michael Dumke	**Version:**	1.0
		Datum:	25.09.2011
Start	18.07.2011		
Ende	05.08.2011	**Dauer:**	15 Tage
Bearbeiter	Michael Dumke		

3.3.1 Theoretischer Entwurf

Satellite attitude control system for demonstration purposes		**AP 3100**	
Titel	Theoretischer Entwurf	**Blatt:**	11 von 22
Verantwortlich	Michael Dumke	**Version:**	1.0
		Datum:	25.09.2011
Start	18.07.2011		
Ende	22.07.2011	**Dauer:**	5 Tage
Bearbeiter	Michael Dumke		

Ziele:

- Entwurf einer theoretischen Lageregelung für den Teststand

Input:

- Klassische Reglerauslegung aus der Literatur

Schnittstellen zu anderen APe:

- AP 1300

Aufgaben:

- Identifikation aller nötigen Parameter, die für die Reglerauslegen benötigt werden (Massen, Trägheitsmomente, Zeitkonstanten der Aktuatoren usw.).
- Verwendung der gefundenen Parameter für die Auslegung einer Regelungsstrategie.

Ergebnisse:

- Allgemeine Algorithmen, welche später in konkreter Form auf dem Onboard Computer implementiert werden können.

3.3.2 Filterung der Sensoren

Satellite attitude control system for demonstration purposes		**AP 3200**	
Titel	Filterung der Sensoren	**Blatt:**	12 von 22
Verantwortlich	Michael Dumke	**Version:**	1.0
		Datum:	25.09.2011
Start	25.07.2011		
Ende	29.07.2011	**Dauer:**	5 Tage
Bearbeiter	Michael Dumke		

Ziele:

- Die Bereitstellung von Sensordaten in einer gut verarbeitbaren Form.

Input:

- Literatur zur Filterung von Messdaten

Schnittstellen zu anderen APe:

- AP 1300

Aufgaben:

- Entwurf bzw. Auswahl von Algorithmen zur Filterung der Sensordaten (inklusive der Sensoren, die den Status der Aktuatoren wiedergeben), falls die Rohdaten nicht verwendet werden können.
- Implementierung der Filter in einer geeigneten Form für den On-board Computer.

Ergebnisse:

- Fertig aufgestellte Filter, welche später auf dem On-board Computer implementiert werden können.

3.3.3 Entwurf von Regelalgorithmen

Satellite attitude control system for demonstration purposes		**AP 3300**	
Titel	Entwurf von Regelalgorithmen	**Blatt:**	13 von 22
Verantwortlich	Michael Dumke	**Version:**	1.0
		Datum:	25.09.2011
Start	01.08.2011		
Ende	05.08.2011	**Dauer:**	5 Tage
Bearbeiter	Michael Dumke		

Ziele:

- Entwurf von Regelalgorithmen für den Lageregelungsteststand.

Input:

- Reglerentwurf auf Basis der realen Parameter des Teststands.

Schnittstellen zu anderen APe:

- AP 1100
- AP 1300
- AP 1400

Aufgaben:

- Entwicklung unterschiedlicher Regler für die Reaktionsschwungräder, die auf unterschiedlichen Anforderungen basieren (Robustheit, Führungsgenauigkeit).

Ergebnisse:

- Fertige Regleralgorithmen, welche später auf dem On-board Computer implementiert werden können.

3.4 Integration und Demonstration

Satellite attitude control system for demonstration purposes		**AP 4000**	
Titel	Integration und Demonstration	**Blatt:**	14 von 22
Verantwortlich	Michael Dumke	**Version:**	1.0
		Datum:	25.09.2011
Start	08.08.2011		
Ende	16.09.2011	**Dauer:**	30 Tage
Bearbeiter	Michael Dumke		

3.4.1 Manöverauswahl

Satellite attitude control system for demonstration purposes		**AP 4100**	
Titel	Manöverauswahl	**Blatt:**	15 von 22
Verantwortlich	Michael Dumke	**Version:**	1.0
		Datum:	25.09.2011
Start	08.08.2011		
Ende	12.08.2011	**Dauer:**	5 Tage
Bearbeiter	Michael Dumke		

Ziele:

- Auflistung der Manöver, die mit dem Teststand simuliert werden können.

Input:

- Dokumentation der Hardware des Teststands.

Schnittstellen zu anderen APe:

- AP 1100
- AP 1400

Aufgaben:

- Auflistung von Beispielmissionen die konkret auf dem Teststand für Demonstrationszwecke verwendet werden können. Insbesondere die Beschränkungen in der Bewegung des Teststands müssen berücksichtigt werden (maximale Drehraten und absolute Auslenkungen)

Ergebnisse:

- Einzelmanöver oder Missionsabschnitte die für eine Demonstration der Teststandhardware verwendbar sind.

3.4.2 Demonstrationsmission

Satellite attitude control system for demonstration purposes		**AP 4200**	
Titel	Demonstrationsmission	**Blatt:**	16 von 22
Verantwortlich	Michael Dumke	**Version:**	1.0
		Datum:	25.09.2011
Start	10.08.2011		
Ende	16.08.2011	**Dauer:**	5 Tage
Bearbeiter	Michael Dumke		

Ziele:

- Szenarien zur Demonstration der Fähigkeiten des Lageregelungsteststandes.

Input:

- Eigene Ideen bzw. Beispiele aus realen Missionen.

Schnittstellen zu anderen APe:

- AP 1100
- AP 1400

Aufgaben:

- Auf der Basis von realen Satellitenmissionen, bzw. selbst entwickelter Manöver, soll eine Demonstrationsmission für den Teststand entworfen werden, die geeignet ist von dem Teststand ausgeführt zu werden.

Ergebnisse:

- Eine konkrete Demonstrationsmission, die die Fähigkeiten des Teststandes so gut und vollständig wie möglich abbilden.

3.4.3 Integration

Satellite attitude control system for demonstration purposes		**AP 4300**	
Titel	Integration	**Blatt:**	17 von 22
Verantwortlich	Michael Dumke	**Version:**	1.0
		Datum:	25.09.2011
Start	17.08.2011		
Ende	16.09.2011	**Dauer:**	23 Tage
Bearbeiter	Michael Dumke		

Ziele:

- Integration der Hard- und Software auf dem Lageregelungsteststand.

Input:

- Lageregelungs- und Teststandhardware
- Die gesamte Interfacesoftware,
- Regelalgorithmen und Filter
- und die Demonstrationsmission bzw. Einzelmanöver.

Schnittstellen zu anderen APe:

- AP 1100
- AP 2100
- AP 4400

Aufgaben:

- Integration der Hardware auf den Teststand.
- Implementierung der Software auf den On-board Computer.
- Durchführung von Tests zur Sicherstellung der grundlegenden Funktionsfähigkeit aller Komponenten.

Ergebnisse:

- Ein voll kontrollierbarer und funktionsfähiger Teststand.

3.4.4 Tests und Demonstration

Satellite attitude control system for demonstration purposes		**AP 4400**	
Titel	Tests und Demonstration	**Blatt:**	18 von 22
Verantwortlich	Michael Dumke	**Version:**	1.0
		Datum:	25.09.2011
Start	17.08.2011		
Ende	16.09.2011	**Dauer:**	23 Tage
Bearbeiter	Michael Dumke		

Ziele:

- Tests zur Gewährleistung der Funktionsfähigkeit und Sicherheit des Lageregelungsteststands.
- Demonstration der Fähigkeiten des Lageregelungsteststands.

Input:

- Soft- und Hardware des Teststands
- Test- und Demonstrationsmanöver

Schnittstellen zu anderen APe:

- AP 1100
- AP 2200
- AP 2300
- AP 4300

Aufgaben:

- Durchführen von Tests zur Beurteilung der vollständigen und sicheren Funktion aller beteiligten Komponenten im Verbund.
- Simulation der entwickelten Demonstrationsszenarien.

Ergebnisse:

- Ein Lageregelungsteststand der sicher und effizient verwendbar ist.

3.5 Dokumentation

Satellite attitude control system for demonstration purposes		**AP 5000**	
Titel	Dokumentation	**Blatt:**	19 von 22
Verantwortlich	Michael Dumke	**Version:**	1.0
		Datum:	25.09.2011
Start	16.05.2011		
Ende	30.09.2011	**Dauer:**	100 Tage
Bearbeiter	Michael Dumke		

3.5.1 Struktur und Inhalt

Satellite attitude control system for demonstration purposes		**AP 5100**	
Titel	Struktur und Inhalt	**Blatt:**	20 von 22
Verantwortlich	Michael Dumke	**Version:**	1.0
		Datum:	25.09.2011
Start	16.05.2011		
Ende	26.09.2011	**Dauer:**	96 Tage
Bearbeiter	Michael Dumke		

Ziele:

- Zusammengefasste Inhalte aller relevanten Diplomandentätigkeiten in einer strukturierten Form.

Input:

- Alle relevanten Geschehnisse währen der Diplomandentätigkeit.

Schnittstellen zu anderen APe:

- alle APe

Aufgaben:

- Aufbau der inhaltlichen Struktur der Diplomarbeit.
- Zusammenschreiben aller wichtigen Tätigkeiten und Auswertungen, inklusive der nötigen Grundlagen.

Ergebnisse:

- Der schriftliche Inhalt der Diplomarbeit in einer geordneten Abfolge.

3.5.2 Grafiken und Diagramme

Satellite attitude control system for demonstration purposes		**AP 5200**	
Titel	Grafiken und Diagramme	**Blatt:**	21 von 22
Verantwortlich	Michael Dumke	**Version:**	1.0
		Datum:	25.09.2011
Start	11.07.2011		
Ende	26.09.2011	**Dauer:**	56 Tage
Bearbeiter	Michael Dumke		

Ziele:

- Erarbeiten aller Diagrammen und Grafiken für die Diplomarbeit in einer einheitlichen Form.

Input:

- Messdaten aus Simulationen
- Skizzen aus Dokumentationen und Literatur

Schnittstellen zu anderen APe:

- alle APe

Aufgaben:

- Erstellung von Diagrammen für die Benutzung im Textsatzsystem LaTeX,
- sowie Skizzen, Ablaufdiagramme und Zeichnungen.

Ergebnisse:

- Der grafische Inhalt der Diplomarbeit in einer ansprechenden Form.

3.5.3 Diplomarbeit

Satellite attitude control system for demonstration purposes		**AP 5300**	
Titel	Diplomarbeit	**Blatt:**	22 von 22
Verantwortlich	Michael Dumke	**Version:**	1.0
		Datum:	25.09.2011
Start	15.08.2011		
Ende	30.09.2011	**Dauer:**	35 Tage
Bearbeiter	Michael Dumke		

Ziele:

- Dokumentation der Tätigkeiten in Form einer Diplomarbeit.

Input:

- Schriftliche Ausarbeitungen der Tätigkeiten.
- Die erarbeiteten visuellen Inhalte (Skizzen, Diagramme usw.).
- Fortgeschrittene Kenntnisse in LaTeX und wichtige Zusatzprogramme für Skizzen, Grafiken und Diagramme.

Schnittstellen zu anderen APe:

- alle APe

Aufgaben:

- Aufbau einer gut verständlichen Dokumentation mit dem Textsatzsystem LaTeX, sowie eine gut anzusehende grafische Aufarbeitung in Skizzen und Diagrammen.

Ergebnisse:

- Eine fertige und qualitativ hochwertige Diplomarbeit.

List of Figures

List of Tables

Glossary

ACS attitude control system. I, 1, 2, 18, 19, 21–27, 35, 51–53, 55, 58, 63, 64, 75, 78–82, 85–87, 90–92, 97–101, 105, 195

AMR anisotropic magnetoresistance. 29–31, 42–44, 99, 194, 196

API application programming interface. 24

ASCII American Standard Code for Information Interchange. 36, 44–47, 196

bit The smallest unit of digital information. A bit has either 1 or 0 as value, therefore a binary numeral system is used to use multiple bits as a representation for numbers. 14, 15, 40, 197

byte Eight bits form one byte. With one byte a number representation of 0 to 255 is possible. 15, 34–36, 38, 40–44, 98, 101, 196, 197

c. m. center of mass. III, VI, 2, 9–12, 14, 18, 21, 26, 33, 49–55, 58, 63, 64, 71, 81, 83, 90, 94, 95, 103, 105, 194

deflection angles The same as the error angles but for an attitude quaternion. For large angles in multiple directions this approach results in a distorted attitude description. 51, 53, 64, 78, 87

DMX digital multiplex. 23, 92

ECEF Earth centered earth fixed. IV, VI, 84, 108

ECI Earth centered inertial. IV, VI, 83–85, 107, 108

EIA-232 A norm for a digital serial connection. 14–16, 24, 33, 44, 92, 94

EIA-422 A norm for a digital serial connection with differential levels for transmission. 16, 17, 24, 31, 33, 39, 42, 94

EIA-485 A norm for a digital serial connection with differential levels for transmission and the capability to connect multiple devices on on serial bus. 24, 33–37, 99

error angles The first three elements of a quaternion are used to visualize the deviation between two attitudes by the doubled inverse sine function of the tree elements: $2 \operatorname{asin}\left(\begin{bmatrix} q_1 & q_2 & q_3 \end{bmatrix}^T\right)$. For small angles the error angles are similar to Euler angles. 52, 80, 81, 87, 97, 197

ESD electrostatic discharge. 93

FACE Facility for Attitude Control Experiments. I, IV, 1, 18, 23, 24, 26, 33, 48, 64, 78, 80, 81, 85, 87, 90, 91, 98, 100, 102, 103, 106, 109, 112, 195

GCC gnu compiler collection. 92

GPS Global Positioning System. 48

HWI hardware interface. 98

IMU inertial measurement unit. 31, 32, 103, 194

LAN local area network. 47, 92

LQR linear quadratic regulator. V, 74, 81, 100, 104, 110, 111

LSB least significant bit. 15

MSB most significant bit. 15

OBC on-board computer. I, 1, 2, 23, 24, 31–34, 38–40, 42, 48, 61, 81, 91–93, 96–99, 101, 103, 104, 194

OS operating system. 24, 41, 99

P proportional. 35, 51

PD proportional-derivative. 49, 50, 52, 72–75, 78–81, 96

PI proportional-integral. 35

PID proportional-integral-derivative. 64, 72–75, 78, 81, 82, 87, 104, 110

quaternion Singularity-free attitude representation. VI, VII, 13, 14, 65, 73, 76–78, 80, 83–86, 97, 99, 100, 197

reaction wheel An actuator to influence the attitude of a satellite by changing the angular momentum of a flywheel. IV, 12, 13, 24–27, 33–38, 49–52, 54–61, 64, 68–71, 73–76, 78, 95–97, 99, 101, 102, 104, 107, 113, 194

RTOS real-time operating system. 24, 92

RTW Real-Time Workshop™. 34, 92, 99, 104

TCP/IP Transmission Control Protocol / Internet Protocol. 47, 48, 94, 98, 99

transformation matrix A mathematical description to change between two different coordinate systems. VII, 14, 65

WLAN wireless local area network. 22, 24, 33, 34, 47, 48, 91–93, 110

Bibliography

[1] Zipfel, Peter H.: *Modeling and Simulation of Aerospace Vehicle Dynamics.* American Institut of Aeronautics and Astronautics, 2000

[2] Wertz, James R. (Hrsg.): *Spacecraft Attitude Determination and Control.* Kluwer Acedemic Publishers, 1978

[3] Wiedeman, C.: *Raumfahrttechnik 1: Raumfahrttechnische Grundlagen*, Institut für Luft- und Raumfahrtsysteme, Technische Universtität Braunschweig, Umdruck zur Vorlesung, 2007

[4] Messerschmid, E. ; Fasoulas, S.: *Raumfahrtsysteme.* Springer, 2008

[5] Markley, F. L.: Attitude Dynamics. In: Wertz, James R. (Hrsg.): *Spacecraft Attitude Determination and Control.* Kluwer Acedemic Publishers, 1978

[6] Fallon, Lawrence: Quaternions. In: Wertz, James R. (Hrsg.): *Spacecraft Attitude Determination and Control.* Kluwer Acedemic Publishers, 1978, S. 758–759

[7] Tandon, Gyanendra K.: Coordinate Transformation. In: Wertz, James R. (Hrsg.): *Spacecraft Attitude Determination and Control.* Kluwer Acedemic Publishers, 1978

[8] Vörsmann, P.: *Regelungstechnik 1*, Institut für Luft- und Raumfahrtsysteme, Technische Universtität Braunschweig, Umdruck zur Vorlesung, 2007

[9] Welch, G. ; Bishop, G.: An Introduction to the Kalman Filter. (2006)

[10] Carvalho, G.: Linear/Extended Continuous-Discrete Kalman Filter Tutorial. (2003)

[11] Lunze, Jan: *Regelungstechnik 1: Systemtheoretische Grundlagen, Analyse und Entwurf einschleifiger Regelungen.* Springer, 2010

[12] Vörsmann, P.: *Flugregelung I*, Institut für Luft- und Raumfahrtsysteme, Technische Universtität Braunschweig, Umdruck zur Vorlesung, 2009

[13] Sidi, Marcel J. ; Rycroft, M. J. (Hrsg.) ; Stengel, R. F. (Hrsg.): *Spacecraft Dynamics and Control.* Cambridge University Press, 2006

[14] Lunze, Jan: *Regelungstechnik 2: Mehrgrößensysteme, Digitale Regelung.* Springer, 2010

[15] Föllinger, Otto: *Regelungstechnik.* Bd. 9. Hüthig, 1994

Technical Documentation

[1] Wiener, A. ; Roemer, S.: Bedienungsanleitung ACS-Teststand (ACS-TS-DLR-ASTRO-TN05). In: *Lageregelungs- und Lagekontroll-Teststand (ACS-TS-A06-0341)* Bd. 1. Albert-Einstein-Str. 12, 12489 Berlin : Astro- und Feinwerktechnik Adlershof GmbH, 2007

[2] Wiener, A. ; Roemer, S.: Funktionsbeschreibung ACS-Teststand (ACS-TS-DLR-ASTRO-TN04). In: *Lageregelungs- und Lagekontroll-Teststand (ACS-TS-A06-0341)* Bd. 1. Albert-Einstein-Str. 12, 12489 Berlin : Astro- und Feinwerktechnik Adlershof GmbH, 2007

[3] Magson GmbH (Hrsg.): *Bedienungsanleitung Magnetfeldsimulationseinrichtung*. 1.0. Carl-Scheele-Str. 14, 12489 Berlin, Germany: Magson GmbH, 2007

[4] Astro- und Feinwerktechnik Adlershof GmbH (Hrsg.): *Lageregelungs- und Lagekontroll-Teststand (ACS-TS-A06-0341) (Teil 3)*. Albert-Einstein-Str. 12, 12489 Berlin, Germany: Astro- und Feinwerktechnik Adlershof GmbH, 2007

[5] RTD Embedded Technologies, Inc. (Hrsg.): *CM17320HR User's Manual Octal RS-232/422/485 PC/104-Plus Module*. E. 103 Innovation Blvd., State College PA 16803-0906, USA: RTD Embedded Technologies, Inc., 2009

[6] RTD Embedded Technologies, Inc. (Hrsg.): *CMA157886 cpuModules User's Manual*. C. 103 Innovation Blvd., State College PA 16803-0906, USA: RTD Embedded Technologies, Inc., 2008

[7] RTD Embedded Technologies, Inc. (Hrsg.): *HPWR104PLUSHR Power supply module User's Manual*. C. 103 Innovation Blvd., State College PA 16803-0906, USA: RTD Embedded Technologies, Inc., 2003

[8] Raschke, Christian: *Spezifikation RW 250-1*. 1. Albert-Einstein-Str. 12, 12489 Berlin, Germany: Astro- und Feinwerktechnik Adlershof GmbH, 2008

[9] Rüd, J.: *µFORS User Manual*. E. Lörracher Straße 18, 79115 Freiburg, Germany: Northrop Grumman LITEF GmbH, 2007

[10] Offterdinger, P.: *Magnetometer Description Operations Manual*. 1. Am Fallturm, 28359 Bremen, Germany: ZARM Technik AG, 2009

[11] iMAR GmbH (Hrsg.): *iIMU-FSAS-CCI/-NCCI Interface Description*. 3.28. Im Reihersbruch 3, 66386 St. Ingbert, Germany: iMAR GmbH, 2011

[12] Trinamic Motion Control GmbH & Co KG (Hrsg.): *TMCM-310 Hardware Manual*. 1.25. Sternstraße 67, 20357 Hamburg, Germany: Trinamic Motion Control GmbH & Co KG

[13] Astro- und Feinwerktechnik Adlershof GmbH (Hrsg.): *Lageregelungs- und Lagekontroll-Teststand (ACS-TS-A06-0341) (Teil 1)*. Albert-Einstein-Str. 12, 12489 Berlin, Germany: Astro- und Feinwerktechnik Adlershof GmbH, 2007

[14] Astro- und Feinwerktechnik Adlershof GmbH (Hrsg.): *Lageregelungs- und Lagekontroll-Teststand (ACS-TS-A06-0341) (Teil 2)*. Albert-Einstein-Str. 12, 12489 Berlin, Germany: Astro- und Feinwerktechnik Adlershof GmbH, 2007

[15] Keithley Instruments, Inc (Hrsg.): *Series 2600 System SourceMeter® Instruments User's Manual*. B. Cleveland, Ohio, USA: Keithley Instruments, Inc, 2007